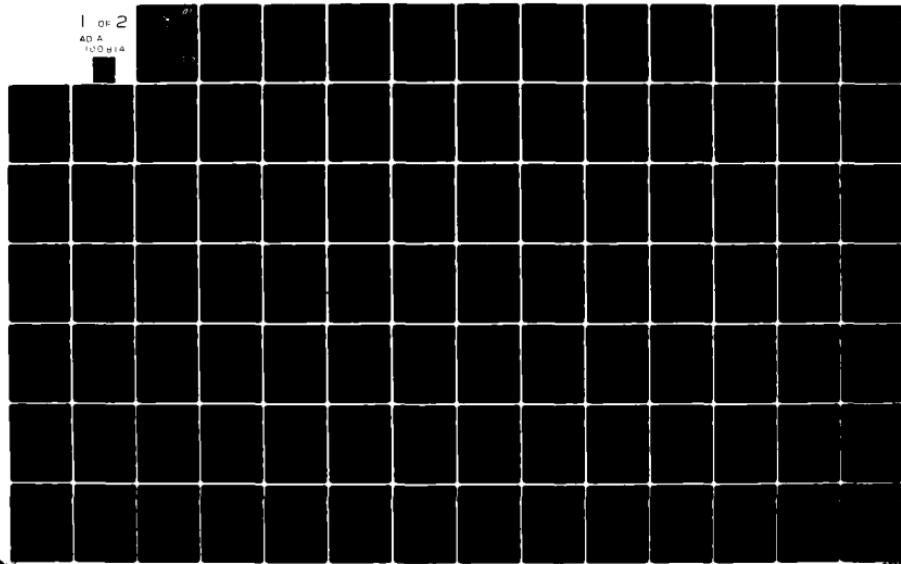


AD-A100 814 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHO0--ETC F/6 9/2
ASGOL--AN ALGOL-STRUCTURED GRAPHICS ORIENTED LANGUAGE. (U)
MAR 81 J D HART
UNCLASSIFIED AFIT/GCS/MA/81M-2

NL

1 of 2
AD A
100 814

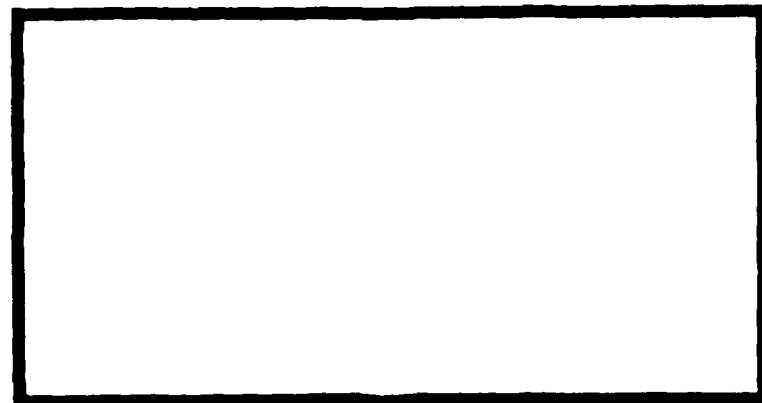


ADA1000014



1

LEVEL II



DTIC
S ELECTED
JUL 1 1981
D

UNITED STATES AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY
Wright-Patterson Air Force Base, Ohio

DTIC FILE COPY

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

81 6 30 066

AFIT/GCS/MA/81M-2

Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	<input type="checkbox"/>
By _____	
Distribution/	
Availability Term	
Avail. Only for	<input type="checkbox"/>
Dist. Special	<input type="checkbox"/>
A	

ASGOL - AN ALGOL-STRUCTURED
GRAPHICS ORIENTED
LANGUAGE

THESIS

AFIT/GCS/MA/81M-2

James D. Hart
2nd Lt USAF

DTIC

1981

ASCOL - AN ALGOL-STRUCTURED
GRAPHICS ORIENTED
LANGUAGE

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

by

James David Mart, B.S.
2nd Lt USAF
Graduate Computer Systems

March 1981

AFIT/GCS/MA/81M-2
AFIT/GCS/MA/81M-2
AFIT/GCS/MA/81M-2

PREFACE

Since my first encounter with computers over five years ago, when I designed a simple basic compiler for the TI-59, I have been fascinated with programming languages. Over the past several years, I have come to believe more and more that programming is more an art form than a science. This can be seen in the almost unlimited number of methods that even the simplest of programs can be implemented. With the introduction of ALGOL and other imitative block structured languages, programming has become even more diversified. However, I hold to the belief that there is one most efficient method for each given program application.

I feel it is not necessary to express the frustrations and anxieties felt when writing a program in a language that is inadequate to effectively perform a task, when the results are not as expected and the error messages do not pinpoint the problems, or when the implementation requires a complete understanding of how the compiler generates the object code. I know I am not alone. However, I view these experiences constructively. It is the inadequacies of the predecessors, as well as the good features, that should facilitate the design of new programming languages. The language designer who does not recognize the faults of previous languages has not considered the full extent to

which his language should apply and is not providing any benefits to the user over other languages. Language design, therefore, is a major engineering task, requiring not only a familiarity with the object code to be generated but a deep understanding of all aspects of programming. The resulting language must efficiently and completely integrate these two into an effective working system.

Since it is true in some cases that these inadequacies are relative only to the specific applications of the language, they are often not discovered until late in the design phase or during the implementation phase of the system. This sometimes requires, as I have discovered, a complete reevaluation of the language to determine the affects the changes may have upon the language as a whole.

The majority of the work for this thesis was applied to the development of the language. The initial request from the sponsors (AFWAL/ACD) was that the software system be supported by PLOT-10, the TEKTRONIX graphics software package. Through the efforts of my thesis advisor, Major Wirth, DISSPLA was shown to have more flexibility and versatility and provide better support in its device independence and plot enhancements than PLOT-10. However, it was not until the end of the third month of research that the change to DISSPLA was finally approved. Much of the design, therefore, that had been structured around PLOT-10 support had to be recomposed to fit that of DISSPLA.

language design continued through the eight month of this thesis. This slowed much of the software system development.

The syntactic error recovery procedure in this system was developed independently. Although there were many references to designing an LR(1) parser skeleton, none of the references give any hint as to how error recovery could be implemented.

I would like to thank at this time Major Michael C. Wirth for his efforts and support in this thesis project for without his help, this document could not have been completed in time. I would also like to thank Capt. Roie Black and Professor Charles Richards for their ideas and contributions to this thesis. I would like to thank Paul Shahady and Hank Maas at AFWAL/ACD for their time in helping me with this project.

Finally, I would like to thank Suzanne Hamel, soon to be Suzanne Hart, for her devotion and support through the trying times of this thesis.

CONTENTS

Preface	ii
List of Figures	v
List of Tables	vii
Abstract	#
I. Introduction	1
II. Requirements and Extensions	5
III. Language Design	7
IV. System Design	18
Main System	19
Scanner Routines	20
Parser Routines	25
Semantic Routines	27
Error Routines	29
Syntactic Error Recovery	30
Symbol Tables	33
V. Recommendations For Extension	39
Tree Structured Language	39
Block Structures	50
Procedures and Functions	52
Graphics Procedures	54
Appendix A BNF code	57
Appendix B System Commands	76
Appendix C Programming Guide	78
Beginner's Guide	78
Program Block Structuring	82
Structure Commands	90
Graphics Instructions	93
Lines and Arrows Instructions	94
Graph Instruction	99
Pie graph	106
Linear and Bar graph	107
Text Instruction	111
Program Enhancement	117
Constants and Variables	117
Arithmetic Expressions	123
Assignment Instructions	133
Page Format Instructions	140
Page Margin Default	143
Graph Instructions	143
Text Instructions	149
String Manipulation	151

List of Figures

FIGURE	PAGE
1 System Interrelationships	19
2 Main System Subroutines	20
3 Scanner Routines	21
4 String Stack Allocation	24
5 Parser Subroutines	26
6 Semantic Subroutines	28
7 Possible Look-ahead States	32
8 Example Symbol Table Structure	34
9 Example Symbol Table Structure	36
10 Axis Variable Record Structure	38
11 Instruction Records	44
12 Instruction Records	45
13 Instruction Records	46
14 Instruction Records	47
15 Node Records	48
16 Node Records	49
17 Procedure Declaration Example	54
18 Program Block Structure	83
19 Page Examples	87
20 Segment Examples	89
21 Margin With Parameters	91
22 Margin Without Parameters	91
23 Margin Error Example	92
24 Text and Graph Combination	94
25 Line Instructions	95
26 Arrow Forms and Locations	96
27 Line and Arrow Instructions	97
28 Lines and Arrows Example	98
29 Linear Axes Definition	101
30 Bar Axes Definition	102
31 Pie Graph Example	107
32 Pie Graph Example	107
33 Bar and Linear Graph Example	109
34 Bar and Linear Graph Example	110
35 Text Example	115
36 Text Example	116
37 Constant and Variable Declarations	121
38 Integer Expression Examples	128
39 Real Expression Examples	129
40 Origin Function Example	132
41 Area Function Example	133

List of Tables

TABLE	PAGE
1 DISSPLA Levels	10
2 Direct Data Graph Coordinames	108
3 Precedence of Operations	125
4 Expression Primaries	126
5 Type Symbols	127
6 Array Variable Assignments	137
7 X Coordinates of Array Components	146
8 X Coordinates for Stacked Graphs	148
9 Shift Character Representation	152
10 Reserved Words and Symbols List	156

ABSTRACT

Graphical methods for visual data display and analysis have had a growing popularity over the past ten years due to their interactive capabilities and the widespread availability of graphic peripheral equipment in which to display the data. However, these tools are often inhibited by language complexity and device-dependence considerations that the user must make. This thesis provides one alternative to these dilemmas.

An ALGOL-structured Graphics Oriented Language (ASGOL) was designed to provide a block-structured format to graphics programs, using several simple instructions to plot linear, bar, and pie graphs. The system was developed around the DISSPLA software package, which generates a device independent file. The DISSPLA post-processing techniques are then used to produce the graphics output to a number of graphics peripherals, including TEKTRONIX video screens and CALCOMP plotters.

In addition, the language provides a text processor technique to draw a variety of character styles and fonts. This text processor can be used to produce entire reports or to document graphs and charts.

I. INTRODUCTION

In situations where business-oriented data changes on a daily or weekly basis, it is becoming increasingly important to display computational analysis of data in various graphical forms that are easily readable by managers. It is not surprising that graphical output is preferred over other forms of data representation since it is an established fact that the human mind can comprehend pictorial information quickly and easily.

The purpose of this thesis was to design a high-level graphics oriented language and to implement this language with an interpreter system. The language was designed to give as much flexibility as possible to the programmer, yet was constrained to have a general purpose instruction set for generating graphs as well as text.

The types of graph forms provided include 2-dimensional linear graph plots (linear-linear, log-linear, linear-log, and log-log axes) with multiple plotting capabilities, bar and stacked bar graphs, and pie graphs. Legends and grids for linear and bar graphs are also available. In addition, the language was designed with several text editor instructions for drawing text anywhere within a page plot. All DISSPLA fonts and styles are available for string manipulation through the language, with variable character height. Also, the size and margins of page plots are adjustable.

The interpreter system was designed to execute on the Aeronautical Systems Division (ASD) Control Data Corporation (CDC) computer, using the Display Integrated Software System and Plotting Language (DISSPLA) package for graphics generation. All plots are

stored in an intermediate device independent file. This file may then be used to display the graphics output through the use of a post-processor to any of a number of devices, including the TEKTRONIX 4010 and 4014 graphics terminals and the CALCOMP plotters. The post-processor is also provided by the DISSPLA package.

The development of the formal language was structured according to the basic application requirements of the system, as set forth by the thesis sponsors (AFWAL/ACD). The Lawrence Livermore Laboratory (LLL) LR Automatic Parser Generator (ref 7) was used to define the language. These tables were then used in an existing LR(1) parser skeleton and combined with the semantics routines of the language and application modules to generate calls to DISSPLA procedures. The complete system was written in FORTRAN, a general purpose high-level language.

ASGOL was designed to provide several advantages over a comparable FORTRAN program with DISSPLA calls. Each of these will be discussed throughout the course of this paper. First, the language is block structured, providing modular program design and syntactic error checks for instructions available only within specific block levels. Second, the instructions were designed to be of a higher level than the DISSPLA calls, i.e., a single ASGOL instruction may invoke more than one DISSPLA calls.

However, several disadvantages have been noted as well. First, since the parsing involves using an LR parser, recovering from a syntactic error is more difficult than the method already designed within most FORTRAN compilers. FORTRAN compilers are weak in their

error diagnostics, which in most cases include only an approximate line number and an error code number.

A second disadvantage is that the ASGOL system does not provide the efficiency in DISSPLA calls that a FORTRAN program does. Several of the calls in an ASGOL program may seem repetitive, but are necessary to contend with the block structure of the language. Various optimizing schemes could be implemented to improve this fault.

The system was originally designed to run interactively with the user. At present this is unfeasible due to the large core memory requirements not only for the system, but also for the DISSPLA package. It is hoped that a later version of this system will be available with interactive capabilities. A set of interactive commands are already provided in the system for such an extension.

The remainder of this paper describes the efforts of this thesis. Chapter II defines the application requirements and the expansions that were applied to the criteria of the system. Chapter III describes the approach of designing the language, using general purpose languages as well as graphics oriented languages as guidelines. In chapter IV, the parts of the system, including how the interpreter was implemented, are defined. Chapter V describes how the language may be expanded to include 3-dimensional and user defined primitive graphics and how non-immediate instructions may be implemented.

Three appendices have been added for additional information of

the system. Appendix A gives the BNF code of the language used to generate the parser tables. Appendix B describes the system commands and how the post-processor is used to produce the graphics output. Finally, appendix C is a programmer's guide to the language.

II. REQUIREMENTS and EXTENSIONS

At the first of every month, AFWAL/ACD, the thesis sponsors, produces a commander's report. This report contains graphs and tables generated from compiled data. Presently, the graphs are drawn using both DISSPLA and PLOT-10 software packages, while most of the tables are typed. By designing a graphics oriented language and developing an interpreter for the language, the majority of the report could be produced from a single program. The initial design of the language was structured around four language requirements set forth by the sponsors. These were that the language should be easily expandable, that graphics data could be read from data files, that the complete program or any part of the program could be run, and that the language have graphics blow-up capabilities.

Expandability can be accomplished by all high-level languages, but as will be seen in the next chapter on language design, block structured languages provide the most efficient and simplest method of implementation.

The data input instructions should be syntactically structured to allow input from either the terminal (for interactive communication) or from an attached data file. Unformatted input and output instructions provide the simplest syntactic constructs. By allowing documentation (comments) to be placed in the data files, the graphics output from an application program can be easily changed by editing the data file rather than the program. This gives better flexibility in how the data is to be presented.

The sponsors wanted a system that would not only generate the

graphs and tables from the complete program, but for sections of the program as well while the AFWAL report is being reviewed. These graphs could then be displayed on a large screen. By defining each of the subtasks of a program (subtasks are defined in appendix C under the section Beginner's Guide) with a unique identifier, this could be accomplished. By including the name of a subtask (or of the program) with the system command RUN (see appendix B), only that block of the program generates graphs.

An extension to this is the page segment blow-up. Several pages of the commander's report contain more than one graph. By specifying the RUN command name as being that of a page segment (where a single graph is defined), DISSPLA calls are limited to only that segment, and the graph is drawn on the complete screen.

Several extensions were made on the initial language design. A text processor instruction could be used to draw text on a page for graph documentation, using any of DISSPLA's character fonts and styles. Several editing instructions could be used to describe the format of the text.

Although the multiple plot, linear graph is the only charting style AFWAL presently uses to produce the commander's report, other basic types should be available in the language, including pie graphs and bar graphs. The language should implement legends for the graphs and have a versatile convention for defining axes.

III. LANGUAGE DESIGN

The approach of graphic languages proposed over the past decade have all been one of three general methods. The most common is called a subroutine package. This design approach, like that used by DISSPLA, is a library of external subroutines or procedures that are called from a general purpose high-level language program. The subroutine method suffers in that all routines are syntactically available anywhere within the program. For this reason, they are not considered "structured language routines". The packages therefore have built into them an internal level structuring through package local variables. These levels insure that all necessary information is presented in an ordered form. If the programmer calls any of these routines out of sequence, errors result.

A less common kind of graphics language is known as language extention. In this approach, new syntactic constructs are added to an existing "host" language. In most cases, a language extension requires that modifications be made to the compiler or even that the compiler be completely rewritten. This presents problems with the portability of the new language dialect (ref 3). The host language chosen is usually a block-structured language due to the "nesting" of blocks in such languages. The most commonly extended language is ALGOL, since the constructs of this language lend themselves easily to the description of graphical structures (ref 9).

The least common method chosen is the design of a new graphics oriented language. A common disadvantage to this approach is that it seldom attracts prospective programmers away from their current

languages. No matter what the deficiencies of the current language, the programmers learn to get around them and often use these deficiencies to their advantage. Graphics oriented languages (and some language extensions) are commonly implemented with an interpreter or precompiler. The source program consists of a sequence of instructions constrained by the syntax of the graphics language. When compiled by the interpreter or precompiler. These instructions are translated into commands for the host language. Although the graphics oriented language poses several design problems, the implementation is basically straight forward. This was the approach taken as the design method for ASGOL.

- In designing a graphics language with the constraints listed in chapter II: that of being an LR(1) language and the requirements set by the thesis sponsors, ten specific parameters were considered important to the language design phase.

1) To establish and enforce the programming conventions that will insure absolute cooperation of the parts: This means that the language should be structured such that the syntax prevents instructions from being called in levels that do not support their implementation. For example, it is impossible to draw a graph until the format of the screen page and the location of the plotting area within the page have been previously defined. Also, the language should assist a programmer in writing large programs when blocks of the program are developed separately and assembled together at a later time. This encourages top-down flowchart design in the development of large or complicated programs.

2) Program structure of the language: ALGOL-like languages provide a high degree of security through the scope and locality associated with block structuring. If a constant or variable is needed for only a particular part of the program, it can be declared to be local to that subtask thus insuring a close association between variables and the instructions that use them. Also, a programmer can be absolutely sure that no other parts of the program will access these variables.

Furthermore, this type of structuring allows dynamic allocation of the names, the types, and the values of variables and constants stored in the symbol tables of the interpreter system. Since the blocks of an ALGOL-like language are always completed in reverse order in which they are entered, the storage area of variables in the symbol tables can be reallocated as soon as they are no longer required.

3) There were two functional requirements of the language that were met in addition to the requirements set by the thesis sponsors: the use of arithmetic operations and string manipulation. In most business situations, the input data used to draw graphs and charts is in the form of raw data requiring computational analysis before they can be graphically presented. In addition, the analysis may represent not only past and present trends, but future trends as well. These trends are often based upon physical mathematical models. By designing arithmetic as well as conditional expressions into the language and applying the standard format for algebraic precedence, the programmer can not only evaluate the input data, but also determine during execution how the data is to be presented.

Since DISSPLA's most significant contribution to graphics display is the string manipulation of character fonts it would have been a mistake not to include several instructions to implement these fonts without inhibiting the versatility of the DISSPLA instructions.

4) DISSPLA Interface Requirements: As stated above, DISSPLA is a subroutine package that implements its own internal level structure. There are four levels in DISSPLA as shown in table 1. With each legal call instruction to the DISSPLA package, a check is made to determine whether the requirements for raising the level of the package have been met. There are additional instructions that must be included to lower the level as well.

The format and structure was carried through to ASGOL for compatibility with DISSPLA. The basic physical structure of DISSPLA is the page which represents a single "image" of one or more objects. Each image is required to have a defined page number, page border, subplot area, physical origin, and an axis system, litigated by the levels shown in table 1.

TABLE 1

DISSPLA Levels	
0	before initialization
1	after initialized page
2	after page border, physical origin, and subplot area defined
3	axis system defined

5) Instruction Control Level: It was inevitable that the hierarchical structure of this language over the host language (DISSPLA) would produce a semantic gap between the two; however, by designing the language with this in mind, the problem was somewhat lessened. The conflict was the inverse relationship between versatility and simplicity. By making the language versatile, the number of instructions in the language become too large for any practical use and therefore require a deep understanding of the constructs, and the language defeats its own purpose. In an opposite light, over-simplification increases the semantic gap and if any type of versatility is available, the instructions become too complex.

6) Error recovery: Many language designers fail to recognize the importance of designing a language with error detection and recovery within the syntax of the language. Instead, it is left to the interpreter or compiler to detect almost all possible programming errors and issue a message for each. This means that the system becomes more complex. Subtle errors may be easily overlooked and the system cannot provide complete error detection. By developing much of the error detection into the syntax of the language, the system is relieved of much of this task, and actually provides better reliability and security against errors.

Since program debugging is often an exaustive and time consuming part of software development, it is also important that when syntactic (as well as semantic) errors are detected, the system provide a detailed description of the error as well as pinpointing

its location. A difficult task in any system is syntactic error recovery, especially for LR parsers. However, it is important that the system recover sufficiently to continue parsing and check as much of a program for further errors as is possible. Error recovery is discussed further in chapter IV.

7) Readability: When the objective of readability by a human being is replaced with readability by the host language, the programmer is forced to write an excessive amount of comments to document a program. When the language is designed to produce good self-documenting code, the user does not have to contend with this difficulty. Reserved words in the language were chosen, therefore, to best describe linguistically what the instructions do.

Also, readability is often sacrificed for writability by providing an unlimited number of default conventions and implicit assumptions in the language, as well as abbreviations of reserved words. These conventions were avoided.

8) Input and output interface: Since the data used to generate graphs and text often come from attached local files, the language was designed with several simple instructions to read data from a number of sequential files. In addition, an output instruction was included to write data to the output file, to the terminal, or to a specified output tape. This provides better debugging capabilities for programs and, more importantly, allows data obtained from computations to be stored for later use.

9) Iterative and conditional instructions: A language whose only primitive structure is sequential execution provides a very limited

implementation format. Almost all languages therefore contain some forms of looping or decision constructs to provide better flexibility.

ASGOL was designed with all four of the primitive structures set forth by Bohm and Jacopini (ref 2). These are the composition, the selection (IF structure), and two iteratives (WHILE and REPEAT structures). In addition, the selection was extended to include the CASE structure and the FOR structure. Since all programs can be written using these primitive structures (ref 2), the GO TO instruction was avoided.

10) Previous experience and familiarity with other languages: Much of the language design was drawn from various existing language structures. The reasons were two-fold. First, an ALGOL-like structure was chosen not only for its nesting level constructs (and ease of expansion), but also for its familiarity to many programmers; and second, much of the language was syntactically structured with existing constructs already widely used.

In particular, five points were considered in the syntax of this language: modularity, open-end parameter formats, column dependency, instruction delimiters, and types of comments.

Modularity is the capability of designing a program in capsules. Each capsule of the program is independent of all others except for the affect it may have upon global variables. Although modularity can be accomplished with almost any high-level language, ALGOL-like languages provide a much more efficient, readable, and most of all expandable method of modularity through their block structure. ASGOL

extends this ability further. By allowing any of the blocks to actually have no instructions at all, it is possible to define blocks early in the program design and add them in later. This also encourages top-down flowchart design, which is an important aspect of any software development.

Open-end parameter formats unfortunately have been used in some graphics languages (ref 11). An open-end parameter format means that not all of the parameters of the instruction have to be specified. When the instruction is interpreted, the parameters not specified at the end of the parameter list are initialized to a predefined default value (by the system or the user).

At first glance, this type of structure seems advantageous, allowing the programmer to define only those parameters that are actually needed. However, problems often surface when debugging a program.

One problem occurs when, for example, a programmer wishes to change only the parameters at the end of the instruction yet does not know what the default values of the parameters are. Most of the languages that use the open-end format allow parameters to be defaulted by a list of parameter delimiters (commonly the comma). If however the programmer does not insert the correct number of delimiters, the system will interpret the declared parameters in the wrong positions (assuming the parameters are of the same type). Naturally, the results will be wrong and the reason for the errors may be hard to determine. Similar situations arise when extra parameter values are inserted or left out mistakenly.

All of these problems describe a specific deficiency of this format: the system cannot compare the number of parameters read to the maximal number of parameters that the instruction allows.

Column dependency: Column dependence occurs when particular columns of each input line (sometimes called a card) are used for a specific purpose. In FORTRAN for example, the first five columns are used as labels, column six is a flag that defines the present line to be a continuation of the previous line, columns seven through 72 are used for instructions, and 73 through 80 (assuming an 80 column format) are not read by the compiler. If a programmer oversteps these boundaries, errors are certain to occur.

Unformatted column structures on the other hand allows better utilization of storage space and provides more flexible indentation for program readability. In addition, blank lines may be inserted to separate sections of the program and a single instruction may occur over any number of lines without setting a flag. (Most FORTRAN compilers allow only 19 continuations for a single instruction.) Furthermore, more than one instruction may be packed on a single line.

When a language such as ALGOL is designed with unformatted column structures, symbols are often used (commonly the semicolon) to establish the end of each instruction. There are good reasons for using instruction delimiters. Delimiters provide better readability, especially when several instructions occur on the same line. However, when delimiters are mandatory, the rules for using them are often confusing. For example, in PASCAL, all instructions within a

block must be followed by a semicolon, except for the last instruction. When blocks are nested, it becomes even more difficult to remember where the delimiters are placed. It was for this reason that an instruction delimiter was avoided in ASGOL. Since all instructions were based upon a common construct of the form:

instruction name (parameter list)

where the instruction name is a single reserved word, it was felt that readability was not impaired.

If the purpose of a programming language is to help a programmer in documentation, the design of a good comment convention must be an important concern (ref 8). There are several designs that have shown to provide little support for documentation. Both comment methods used by ALGOL-60 were eliminated. The first method allows placing comments between an END and the next semicolon, END, or ELSE found on the input string. This convention can prove disasterous if the end of the comment delimiter is omitted. The second ALGOL-60 method allows placing comments between the word COMMENT and a semicolon. The word COMMENT unfortunately occupies space that could be better utilized (ref 8).

Another method eliminated due to its column dependence, was one in which a delimiter is placed in a specific column to define an end-of-line. Characters after the delimiter are not scanned as text. This is the method used by some assembly languages and by FORTRAN.

Two methods remained in consideration. The first is similar to

the previous case except that there is no column dependence upon the delimiter. The second convention is the use of special brackets to enclose comments. Both of these conventions were implemented by ASGOL. The delimiter in the first case was chosen to be a double character set (--), since this reduces the chances of a mispunch over a single character delimiter. The special bracket convention was taken from PASCAL: the bracket head /* and tail */.

IV SYSTEM DESIGN

Once the initial design phase of the language (the first of five) was completed and the language was written in LR(1) BNF code, the parsing tables were generated by the LR(1) language analyzer. A description of these tables is given along with the BNF code used to generate the LR tables in appendix A. It is not necessary to understand how the parsing procedure works unless the language is expanded to include other constructs. Semantic changes to the language can be made directly in the system's semantic routines. System expansion is discussed in chapter V.

The parsing tables are used by the system through the common block called TABCOM. This common block is only available to those routines that require them.

The system's subroutines are combined into five separate groups: the main system, the scanner routines, the parser routines, the semantic routines and application modules, and the error routines. The interrelationship of these groups are shown in figure 1.

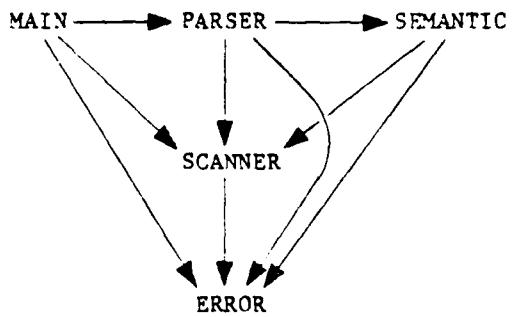


FIGURE 1 System Interrelationships

The last section discusses the implementation of the symbol tables and the common blocks associated with the storage of symbol information for the entries.

Main System:

When the system begins execution, it sends a welcome message to the terminal and issues a ready prompt "COMMAND=". (The system was designed with the intention that later versions of the system would run interactively with the user.) A ready prompt is issued every time the system is ready to accept another command from the terminal.

The system commands are entered from the input file (or the terminal) and the system retrieves the tokens of each command

individually by calling the scanner routines, which return the entry number of the command token in the vocabulary. A complete description of the system commands is listed in appendix B. Commas are used between each token to flag the program that additional tokens remain on the input line. If the user does not supply the system with all needed information required to perform the command, the system will prompt the user for the information needed. Since the system at present does not run interactively, it is recommended that all information be supplied on each input line. Figure 2 shows the routines associated with the main system.

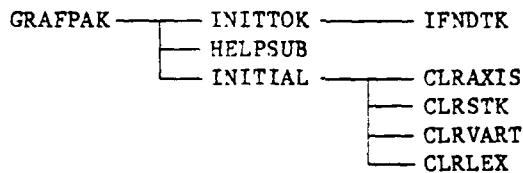


FIGURE 2 Main System Subroutines

When the RUN command is detected, the main system calls the initializing routines which set all common block variables needed for the grammar parse to initial values. The subtask name specified in the RUN command is entered into the symbol tables for reference by the semantics routines and a call is made to DISSPLA to create an intermediate plot file called PLFILE. The parser routines are then called to begin the parsing procedure.

Scanner Routines:

The scanner routines (also called the lexical analyzer) are called to retrieve tokens from three locations: command tokens from

tape 5 when called by the main system; language tokens from the program input file (tape 7) when called by the parsing routines; and data tokens from one of the data input files when called by the semantic routines. All data used by the system are read through the scanner routines.

After each line of the input has been scanned, GETLIN is called. This subroutine reads the next input line from one of the input files and stores these characters in a line buffer. Two pointers are used with the line buffer: one to point to the next unused character in the line buffer; the other to point to the last character in the line buffer. When GETLIN is called to read a new text line from the program input file, a check is made to determine if an error had been detected on the previous line. If so, a point "^" is written to the program output file (tape 10) in the appropriate columns where each error occurred. The line read is then written to the program output file. Figure 3 shows the subroutines associated with the scanner routines.

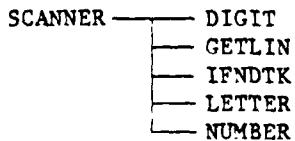


FIGURE 3 Scanner Subroutines

When the scanner routines are called, the next token is determined by the following steps:

- 1) remove blanks and comments from the text;

2) detect for multicharacter tokens, i.e., reserved words and identifiers

3) detect for special symbols, e.g., "=" and "=>";

4) detect for numbers, i.e., integers and reals;

5) detect for character strings; and

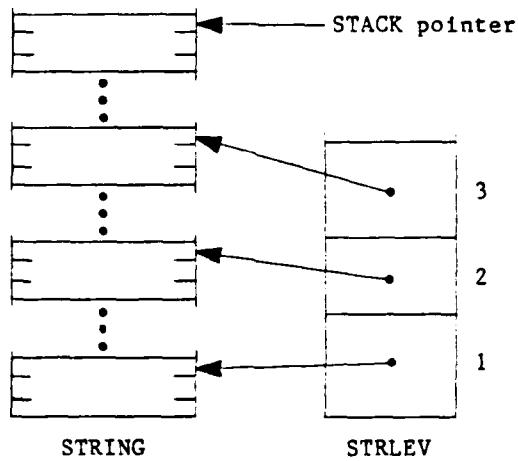
6) issue an error message that the token read is an illegal token, then scan for another token. When the token being scanned has been identified, it is returned through SCANNER's single parameter.

For reserved words and special symbols, the parameter contains the pointer into the vocabulary of the token read. For an identifier, the name is placed in a symbol string (SYMSTR). For numbers, the digits are read sequentially and translated into the internal machine representation, and the values are passed back through either of two lexical common variables IVAL or RVAL.

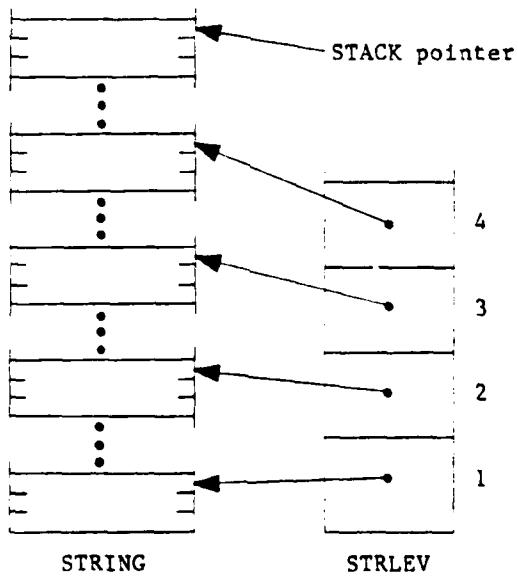
Since the system was written in FORTRAN which requires that memory be declared statically, several methods were designed to provide better efficiency of storage space. The manipulation of character strings was one such case. A string stack was designed to store character strings in a dynamic fashion. Character strings are read into the stack starting at the stack top by "pushing" each character as it is read. After the complete character string has been pushed, SSTART returns the pointer to the top of the stack prior to inserting the character string, and SLNGTH returns the number of characters pushed onto the stack (the number of cells allocated to the string.)

STRLEV is a single-dimensional array that contains the pointers

to the stack top for each lexical level the system is in. Each time a new level is entered, LEXICAL is incremented and the pointer to the stack top is pushed onto STRLEV. This simulates dynamic allocation for each lexical level. As characters are read, they are pushed onto the stack and the stack pointer is changed to point at the top of the stack. However, since the stack pointer of the previous level has been saved, at the end of a block, LEXICAL is decremented and the stack pointer is "popped" from the STRLEV stack. (There are no instructions in the stack pop; it is performed implicitly when LEXICAL is decremented.) All of the characters pushed onto the stack in the level completed are no longer accessible. This simulates a deallocation of the STRING array. Figure 4 illustrates allocation and deallocation of character strings.



(a)



(b)

FIGURE 4 String Stack Allocation

In figure 4(a), the system parser enters the fourth level after which a block of characters were pushed. In figure 4(b) another level has been entered and a block of characters were pushed onto the stack within this level. When level five completes, the string stack returns to the state in figure 4(a).

Parser Routines:

The parser routines, combined with the LR(1) tables generated from the BNF language description, form the LR(1) parser package. The parser skeleton (the LR(1) parser and the scanner routines) of the system was provided by a previous graduate student at AFIT (ref 10) who obtained them from the Lawrence Livermore Laboratory (ref 7). The skeleton was modified to fit the specific applications of the ASGOL system.

The LR(1) parser works as a push-down deterministic finite automatum. Four of the five properties of the 5-tuple are provided by the tables in the TABCOM common block:

- 1) a token vocabulary,
- 2) a finite alphabet of elements called states,
- 3) an initial state, and
- 4) a final state.

The fifth is a set of functions that determine the next configuration state of the parser given the current state and a

single look-ahead token (combined to be called a handle). LR(1) means that only one look-ahead token is required to uniquely determine the next state. The push-down was implemented with a parser stack and developed so that each stack location would contain all information needed for each configuration state.

The LR(1) parser generates a left-right bottom-up parsing procedure of the input program (called a grammar). The bottom-up technique is to start with the input tokens (passed by the scanner) and try to reduce them into non-terminal symbols. Figure 5 shows the subroutines associated with the parser routines.

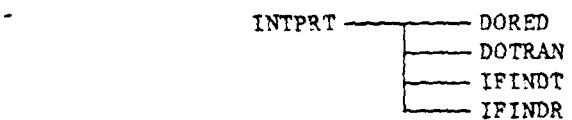


FIGURE 5 Parser Subroutines

The LR(1) parser procedure is implemented through a FORTRAN constructed compilation DO loop. Each loop corresponds to a single canonical parse step. An exit is made from the loop on one of two conditions: when the current state of the parser is the final state or when a syntax or syntactic error is detected and recovery cannot be made. (Error recovery is discussed later.)

Within the compilation loop, one of three sequential actions are performed for each iteration. The first action is a reduction attempt given the handles and the current look-ahead symbol. A reduction is performed by popping from the parser stack a specified number of stack locations. The actual number corresponds with the

length of the syntax sentence being reduced. The semantic routines are called and the production number is passed as the parameter. Upon return, the new configuration state is found and the state, the look-ahead token, and the information from the semantic routines are stored into the semantic stack top location.

The second action is a transition attempt. A transition is accomplished by pushing the new state (determined by the handles and current look-ahead token) along with the look-ahead token, on to the top of the semantic stack.

If both the reduction and transition attempts fail, a syntactic error has occurred and the error routines are called. A syntactic error occurs when an illegal look-ahead token has been scanned. A token is illegal when it is not one of the tokens acceptable to perform a reduction or transition. If the error routines recover, the parsing continues; otherwise, system control returns to the main system.

Semantic Routines:

The semantic routines, along with the application modules, are used to generate the host language (DISSPLA) calls. These routines also describe the semantics of the language. A semantic routine contains the set of instructions necessary to perform the semantics of one language syntax sentence. All of these routines have been written into a single FORTRAN subroutine called SMANTK. A set of GOTO's at the head of the subroutine specify which routine is to be implemented with each SMANTK call.

Once a program has been parsed enough to draw a graph or text, the semantic routines call the application modules that perform the DISSPLA calls. The application modules have not at this time been fully developed. Figure 6 shows the subroutines associated with the semantic routines.

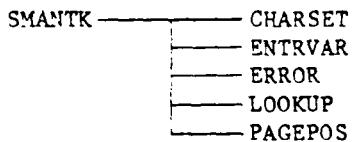


FIGURE 6 Semantic Subroutines

Most of the routines describe the semantics of the language by performing computations or changing and manipulating data stored in the semantic stack and symbol table common blocks. (Several of the routines contain no instructions at all.) The information needed for each routine can be obtained from the parser stack above the stack top. This information is then reduced and stored into the location at the semantic stack top.

Several of the parser routines have not been implemented. Most of these deal with the iterative and conditional constructs of the language. These constructs involve non-immediate executable instructions. For example, consider a simplification of the language **WHILE** construct:

```
WHILE <expression> DO <block list> END WHILE
```

Since this construct may involve a conditional loop of more than one instruction (or even subtasks), the instructions must be retained in their original form before they are reduced to <block list> and the while expression can be evaluated. The information needed for the complete <block list> must be stored in a single semantic stack entry. One method for accomplishing this is described in the next chapter.

Error Routines:

During the interpretation of a program, four types of errors may occur: system errors, syntactic errors, semantic errors, and warnings. Each of these affect the system in different ways and are handled differently.

System errors occur when the program exceeds the dynamic allocation limits of the system stacks. These errors include a parser stack overflow, a STRING stack overflow, a variable table overflow, and an axis table overflow. Since the parser or the semantic routines can no longer effectively interpret the program, the parser writes the error diagnostics to the output file and returns to the main system.

Syntactic errors occur when an illegal token has been scanned. Syntactic error recover is discussed later.

Semantic errors and warnings occur in the execution of the semantic routines. Although both of these do not affect the parsing procedure, when a semantic error is detected, the DISSPLA package

calls are discontinued. Warnings recover sufficiently enough to continue without affecting the DISSPLA calls.

After the program has been parsed, error diagnostics are written to the output file giving the location (line number and column) and a short description of every error detected. Presently, the system supports over 100 semantic error and 68 warning checks in the semantic routines as well as all system and syntactic errors.

Syntactic Error Recovery:

The LR(1) grammar parser generates a syntactic error whenever the current look-ahead token is neither a legal token for a reduction nor for a transition. To determine which of the two can be performed at the current configuration state (NOWSTA) where the error occurred, a check can be made with the one-dimensional arrays FRED and FTRN generated by the language analyzer. A reduction is possible if FRED(NOWSTA) is less than FRED(NOWSTA+1); a transition is possible if FTRN(NOWSTA) is less than FTRN(NOWSTA+1). One of these checks will always be true for each state in an LR(1) language (except for the final state) since each state must provide a path to another. Furthermore, it can be assumed that the cause of the error is either:

- 1) a sequence of one or more illegal tokens have been added to the program,
- 2) the present illegal token scanned has replaced a legal token, or
- 3) a sequence of one or more legal tokens have been omitted from the program.

Syntactic error recovery occurs in two steps. First, if a reduction is possible at NOWSTA, a reduction error recovery attempt is made. If a reduction is not possible or if the reduction attempt fails (several situations arise where both reductions and transitions are possible), a transition error recovery attempt is made. If both attempts fail, the system is unable to recover from the error.

The implementation approach for both the reduction and transition attempts are very similar. They differ only in determining all of the possible states that may occur after NOWSTA, the configuration state when the syntactic error was detected. Given NOWSTA, it is possible to determine all of the next legal tokens that are acceptable in the input string. These tokens can then be compared with a set of look-ahead tokens scanned from the input. However, as can be seen in figure 7 for N legal look-ahead tokens, the number of searches increase exponentially as the level N increases linearly. The circles represent the configuration states that occur given the state at the previous level and a legal token.

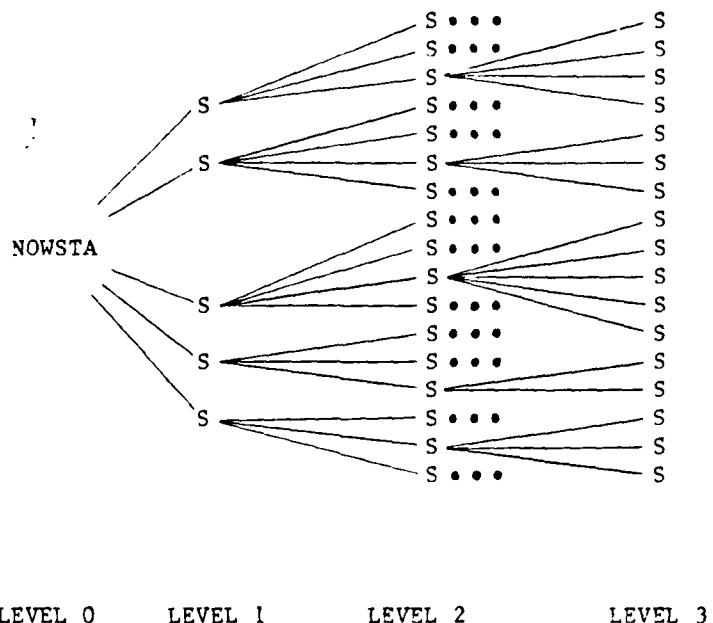


FIGURE 7 Possible Look-ahead States

Since the complexity of the implementation also increases with the number of searches required for N levels (i.e., at each new state level, all reduction and transition checks must be made), the reduction and transition attempts were limited to three look-ahead tokens (the illegal token and the next two tokens on the input string). This was chosen as a compromise between recovery reliability

and the amount of computer time and memory necessary for all searches.

Symbol Tables:

Checking the correctness of a program's semantics and performing the semantic actions requires knowledge of all identifier attributes. These identifiers are the constants, variables, and subtask names used in the source program. The attributes of constants and variables include the name, the basic type, the array dimensions, the lexical level, and the value(s) of each identifier. For subtask names, the attributes include the name and the subtask type (PROGRAM, SECTION, PAGE, SEGMENT, or RUN identifier).

A record to contain the attributes of an identifier is allocated within the semantic routines when the identifier is declared. A hashing function applied to the identifier name returns a hash address (between 1 and 50) and ENTRYPTR(address) becomes the pointer into the new record. The size of the bucket, ENTRYPTR, was chosen for conservation of memory with a minimum number of name collisions. Collisions are discussed below. Figure 8 illustrates how two identifiers AAA and BBB are entered into the table. The hashing method provides an efficient method of searching for identifiers. If, for example, a search was to be made for AAA, the hashing function will always return the same address.

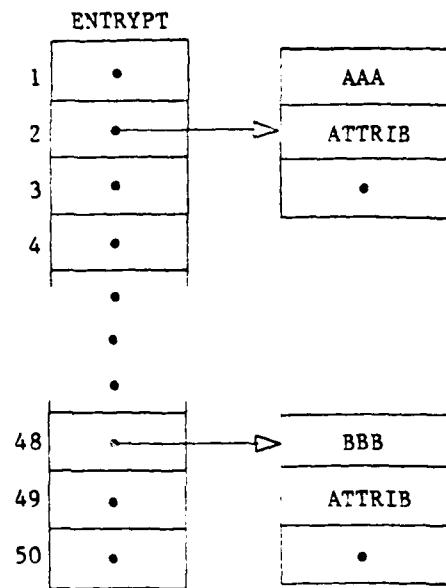


FIGURE 8 Example Symbol Table Structure

A special case occurs when two identifiers hash to the same address (called a collision). This will always be the case for identifiers with the same name but declared in two different levels. Therefore, a chaining method was implemented. Each table record has a pointer attribute that points to the next identifier at the same hashing address.

When a collision occurs, the new record is inserted at the head of the chain. There are two reasons for this. First, when the table is searched for a specific entry, the hashing function returns the

address of the head of the chain. Each entry node along the chain is compared with the entry name being searched for. The rule for finding the correct declaration of an identifier is to look in the current lexical level or block of the program and proceed outward until it is found. Although two identifiers with the same name hash to the same address, the search ends when the first identifier is encountered, which is always the one in the closest surrounding block. Any other identifiers with the same name therefore are inaccessible to the system.

The second reason for inserting records at the head of the chain involves the deallocation of records each time a block is exited. Since blocks are exited in reverse order in which they are entered, all identifiers declared within the block are at the head of the chains. By searching down the bucket, a check can be made into the head of each chain. If the lexical level attribute of the first record of the chains corresponds to the block level being exited, the record is removed and the next entry of the chain is placed at the head. The record is deallocated and the comparison is repeated until all identifiers declared within the block have been removed. Consider, for example, an identifier CCC declared within a block local to the block AAA and BBB were declared in, as in figure 8. If the hashing function "maps" the name CCC into the same address as BBB, the results are as shown in figure 9. When the block that CCC was declared within is completed, the name is removed and the structure returns to that shown in figure 8.

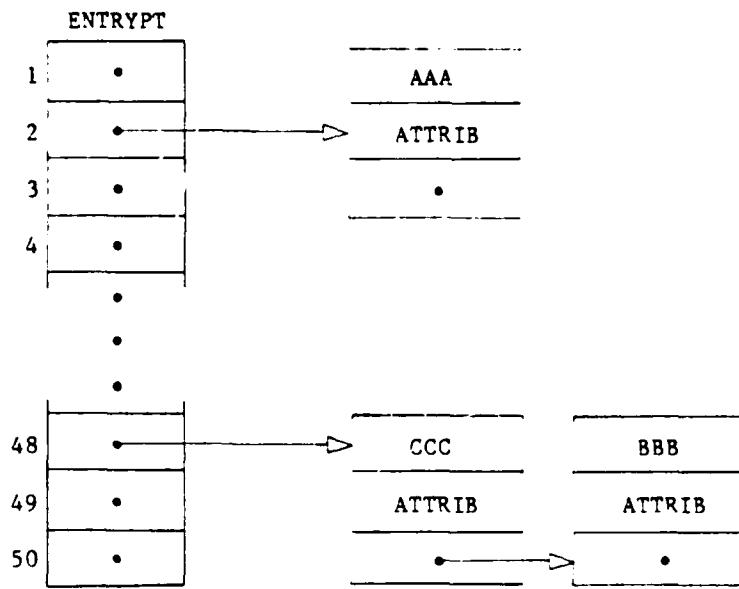


FIGURE 9 Example Symbol Table Structure

When an identifier is declared to be one of the basic types integer, real, boolean, or character, or a unit variable, it is allocated memory within the value stack (VALSTK) (one location for simple constants and variables and N locations for array variables, where N equals the product of the declared dimensions). A pointer into the first location of the value stack allocated is stored in VALPTR, one of the attributes for all constants and variables. The value stack comprises two single-dimensional arrays: the first for storing the values of the constants and variables, and the second for

determining whether the variables are defined or not. When a block is completed, memory is deallocated from the value stack and the second array is reset to undefined on the deallocated locations.

For constants and variables declared to be of type string (of defined length), memory is allocated in the STRING stack. Each string identifier's attribute STRSTART contains a pointer to the top of the stack before the allocation occurred and STRLNG contains the length of the string identifier. The STRLEV bucket is updated at each allocation in the same manner as that used to allocate storage for character strings read in the scanner routines. The record attribute NEXTCH serves two purposes. First, it always points to the 'next character to be printed to the screen. (see the use of the function POINTER described in appendix C under expressions.) Second, when a record is allocated for an identifier, NEXTCH is set to 0. This value is set to 1 only after the string identifier is defined. For string constants, this will always occur within the declaration.

For axis variables, two records are allocated. The first is the type already described: the variable record. The second is an axis record which contains all six of the attributes needed to define an axis: the title, the axis type, the min and max points, delta, and ticks. The variable record is inserted at the head of the chain and its attribute AXISPTR contains the pointer into the axis record, as shown in figure 10.

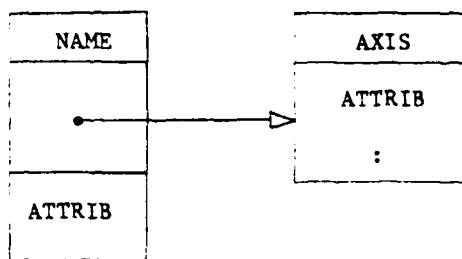


FIGURE 10 Axis Variable Record Structure

CHAPTER V

This chapter is divided into two basic sections. The first describes a method of executing the iterative and conditional instructions of the language and how this method may be applied to interactive capabilities. The second section explains some ideas on extending the language to include other constructs.

As stated in the previous chapter, for the iterative and conditional instructions to be executed, the instructions (as well as declarations for conditionals) within the instruction block must be saved in their original unexecuted form until the iterative or conditional instruction has been completely reduced. For example, consider an IF <true branch> containing the declaration and use of a variable. The instruction within the instruction block cannot evaluate the variable when it is encountered since it has not been entered into the symbol table, i.e., the declaration has not been executed. Instead, the variable name must be retained for later evaluation. The described method below should be considered only as one possible method. The description of the records used for this method are not guaranteed to be completely correct. Some modifications to this design may be necessary.

The design approach involves two steps. The first is the design of an intermediate tree structured language. The second step is replacing the semantic routines of the present system with a routine that builds a tree from the input text until a point is reached where the instructions can be interpreted.

A tree structured language was chosen because this approach

provides a way of not only retaining the order of instructions but retaining the block structure of the language as well. In addition, the LR parsing and the basic nesting constructs of the language simplify building the tree.

There were two record types designed to implement the tree structure: the instruction record and the node record. Two record types were chosen due to the common grouping of the language syntax. The number of record components for each type provides efficient storage utilization. Furthermore, each record component was designed with as much commonality as possible to reduce ambiguity. The first component of the record types was chosen to contain an integer mnemonic to specify the constructs of the record.

In addition to these records, two array stacks are implemented with the tree structure. One stack, noted in this chapter as CSTR is used to store all character strings read from the program text. This stack is different from the lexical stack STRING. Where CSTR stores all strings read while parsing not immediately executable instructions, STRING stores only the characters within the lexical blocks being executed.

Figure 11 through 14 show the structure of each mnemonic type in the instruction record. All component names beginning with the pointer symbol (#) represent pointers into the record names specified. All other names are interpreted as actual values. Shaded record components represent those not recognized by the construct. The tables following the record figures describe one method of interpreting these values.

The last component of most instruction records are pointers to the next instruction record (IR) of the sequence of instructions. The SECTION, PAGE, and SEGMENT record constructs were designed to retain the block structure of the language. The PROGRAM construct will be explained later.

The node records are used to construct all expressions, specify titles, and store data constants read from the input text file. All expressions are defined with the EXPRESSION node record at the tree root. The VARIABLE and FUNCTION node records are the end-vertices of the expression trees. All other expression nodes are defined with the OPERATION node record. The mnemonics of the node records are shown in figure 15 and 16.

The CSTR stack is a one-dimensional array. The stack is implemented completely with the CS node record. Each time a new string has been pushed onto CSTR, a new CS record is allocated, and the pointer to the first character in the string and the string length are stored. CSTRSP is the pointer to the top of the CSTR stack.

Like the CSTR stack, the Identifier Stack Entries (ISE) stack is implemented only while parsing not immediately executable instructions. The ISE stack contains all identifiers encountered within the declaration blocks of the input text. Each stack entry holds the complete name's character string and a pointer to the previous stack entry for the current lexical level. For the first identifier of a subtask (or block), the pointer links the current block entries to the last entry of the immediate outer contour. Each

time the identifier is used within the block, the stack is searched and the entry pointer (#ISE) is returned. It is possible to allocate an entry each time an identifier is encountered; however, this does not provide an efficient use of storage space. If an identifier is not found within the ISE stack, the symbol table is searched.

In implementing an intermediate tree language, each semantic routine must be replaced with a procedure that allocates memory in both record types and both stacks while parsing the input text; then, after storing the information within the records and stacks, links the records into a tree structure. Once the parsing has reached a point where the tree can be interpreted, the tree language interpreter (TLI) is called, and a pointer into the tree root is passed.

One of the major disadvantages of the present system is that it is too large for any practical interactive capabilities. With the PROGRAM instruction record, it is possible to separate the system into two parts; the first part being a "compiler" that using the LR(1) parsing procedures, generates a tree from the complete program text (source code), the second part being the tree language interpreter. After the parsing is completed, the compiled tree language (including the two stacks and both record types) can be written to a tape, to be later read by the tree language interpreter.

The "compiler" portion of the system would include the parsing routines (with the tables generated by the LR(1) language analyzer), the scanner routines, the semantic routines that generate the tree, and a portion of the error routines. The types of errors that the

compiler should detect are system stack overflows, syntactic errors, and all compilation semantic errors and warnings. This relieves much of the run-time error detection necessary during interpretation of the tree language.

The "interpreter" portion of the system would include the main system (which accepts all system commands), the scanner routines, the TLI routines, and the application modules. The scanner routines are necessary for the unformatted input and output instructions of the language.

PROGRAM	1	#ISE	#IR			
SECTION	2	#ISE	#IR	•--	NEXT SECTION	
PAGE	3	#ISE	#IR	•--	NEXT PAGE	
	4	DIR	MAR	#EXP	LOC	
SEGMENT	5	#ISE	#IR	#UVL	•--	NEXT SEGMENT
CONST#DECL.	6	#ISEL	#EXP		#IR	
VAR#DECL.	7	#ISEL	TYP	#EXPL	#IR	
BINDING	8		#UV		#IR	
GRACE	9		#UV		#IR	
BORDER	10		#UV	#UV	#IR	
MARGIN	11		#UV	#UV	#IR	
FRAME	12		#EXP		#IR	

FIGURE 11 Instruction Records

SET (BASIC) 13 #VAR TYPE #EXP #IR

SET (AXIS) 13 #VAR 5 #AXS #IR

SET (UNIT) 13 #VAR 6 #UV #IR

SET (STRING) 13 #VAR 7 #EXP #IR

INPUT 14 #ISE TAPE // / #IR

14 #ISE 0 #DL #IR

OUTPUT 15 #ISE TAPE #IR

CHANGE 16 // / FROM TO #IR

HEIGHT 17 // / #UV // / #IR

IF 18 #EXP #EXP #EXP #IR

CASE HEAD 19 #VAR / / / #IR

CASE SEQUENCE (CS) 20 #CON #IR #CS

FIGURE 12 Instruction Records

WHILE	21	#EXP	#IR	1	#IR
REPEAT	22	#EXP	#IR		#IR
FOR	23	#VAR	#IR	•	#IR
FOR LIST	24	#EXP	U-D	#EXP	#EXP
DRAW LINE	25	1	1	#UVL	#IR
DRAW ARROW	25	2	INT	#UVL	#IR
	25	3	#VAR	#UVL	#IR
AXIS	26	X-Y	1	#AXS	#IR
	26	X-Y	2	#ISE	#IR
GRID	27	1	#EXP	#EXP	#IR
LEGEND	28	#TIT	STACK	FRAME	1
LEGEND	28	#TIT	LOC	#TITL	#IR

FIGURE 13 Instruction Records

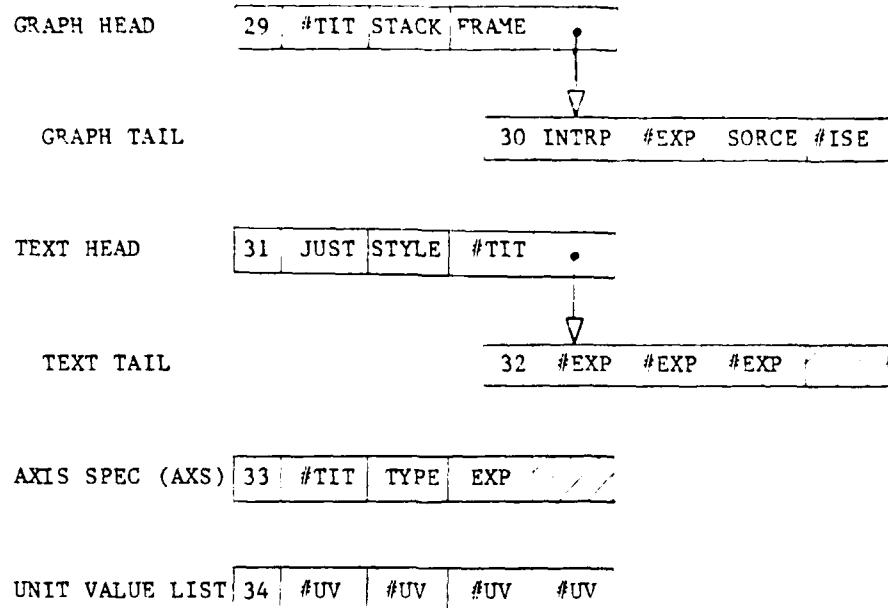


FIGURE 14 Instruction Records

EXPRESSION (EXP) 1 #NR | 0

EXPRESSION LIST (EXPL) 1 #NR | •-- NEXT EXP NR

VARIABLE (VAR)

BASIC 2 #ISE #EXPL

UNIT 3 #ISE ~~✓✓✓~~

AXIS 4 #ISE ~~✓✓✓~~

STRING 5 #ISE ✓✓

FIGURE 15 Node Records

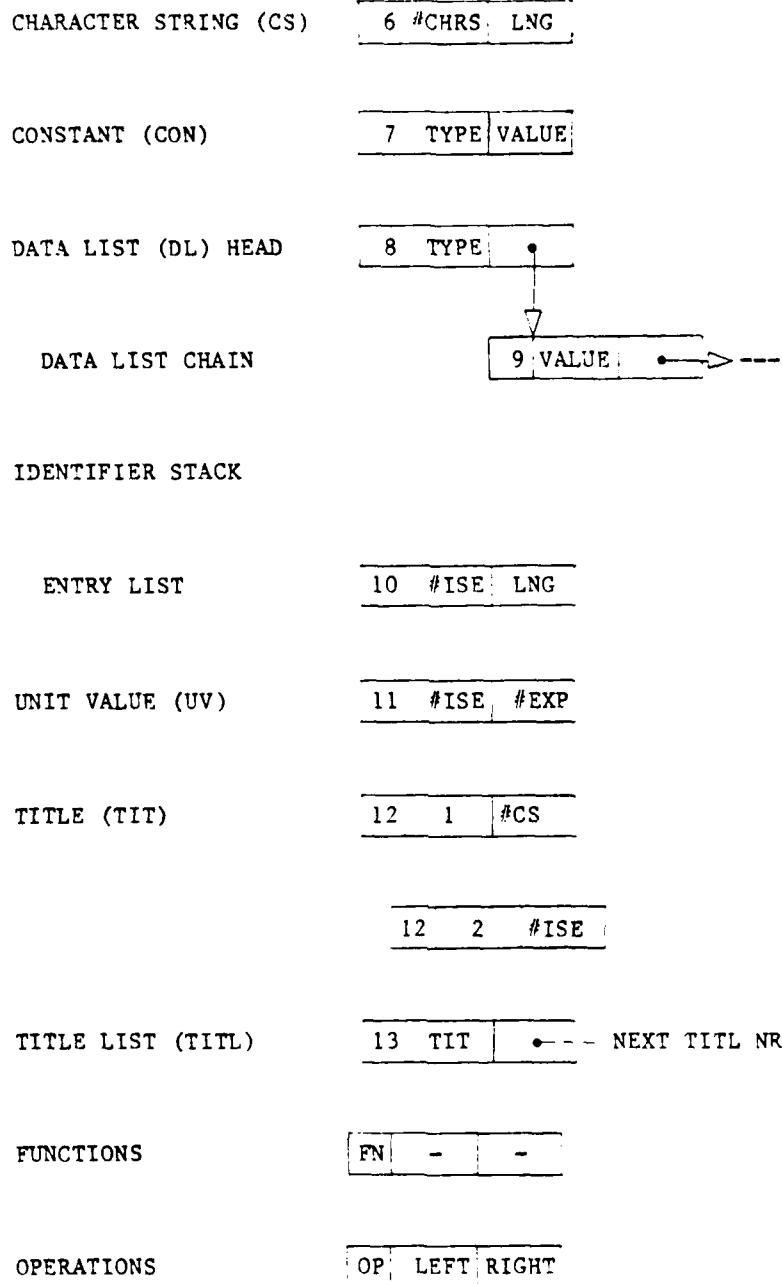


FIGURE 16 Node Records

Although much time and effort went into the language design, many syntax constructs were not implemented in this version of ASGOL. This section describes four important concepts in adding to or changing the syntax to provide a more efficient and flexible language. The first two constructs deal with features that should be available for any block structured language: the definition of program blocks, and procedures and functions. The last two describe methods of extending the graphics capabilities of the language to include and user defined 2- and 3-dimensional primitive objects.

Block Structures:

True block structured programming languages such as ALGOL allow the definition of program blocks anywhere where instruction statements (the BNF code in appendix A describes these as structure commands) are allowed. Within these blocks, variables and constants needed only for a particular part of a subtask (the difference between blocks and subtasks is defined below) can be declared. This description correlates with the declaration and use of constants and variables within subtasks and provides the same advantages: close association between the variables and the code which uses them.

The syntax sentences of a block may be defined as:

```
<section block> ::= <block head> <declaration list>
                         <section structure command list>
                         <block end>
<page block> ::= <block head> <declaration list>
                         <page structure command list>
```

```
        <block end>

<segment block> ::= <block head> <declaration list>
                    <segment structure command list>

<block head> ::= BLOCK

<block end> ::= END BLOCK
```

The syntax of the block definitions were broken into the three major categories SECTION, PAGE, and SEGMENT for the same reason block lists of the iteratives and conditional instructions were divided. These constructs provide syntactic error detection of the structure commands available within the block, e.g., page margin instructions are syntactically not allowed with a page or segment subtask. The block definitions may now be included within the structure commands as follows.

```
<section structure command> ::= <section block>

<page structure command> ::= <page block>

<segment structure command> ::= <segment block>
```

With the syntax definitions shown above, it is impossible to define a subtask within a block. This prevents confusing blocks with subtasks. Where subtasks describe the graphics representation level of a program (e.g., a page subtask defines the structure of a single page to be drawn), instruction blocks provide only the capability of declaring a group of variables local to the structure commands that require them. Also, while subtasks are given names, blocks are not.

Procedures and Functions:

One of the most important constructs of any high-level language is the declaration and use of functions and procedures within a program. When used, these constructs produce compact source code. Just as important as the description of procedures and functions are the definitions of their parameters and the method in which they are passed both into and out of the procedures.

To distinguish between the declaration and invocation of procedures and functions, procedures should be declared within the declaration list of a block or subtask. The syntactic structure could be as follows.

```
<declaration list> ::= DECLARE
    <constant declaration list>
    <variable declaration list>
    <procedure declaration list>
    END DECLARE
<procedure declaration list> ::= <null>
    <procedure declaration list>
    <procedure declaration>
    / <procedure declaration list>
        <function declaration>
<procedure declaration> ::= <procedure head>
    <procedure body>
    <procedure end>
<procedure head> ::= PROCEDURE <identifier>
    <parameter list>
<procedure body> ::= <section body>
    / <page body>
    / <segment body>
<procedure end> ::= END PROCEDURE <identifier>
<function declaration> ::= <function head>
    <function body>
    <function end>
<function head> ::= FUNCTION <identifier> : <type>
    <parameter list>
<function body> ::= <section body>
    / <page body>
    / <segment body>
<function end> ::= END FUNCTION <identifier>
<parameter list> ::= <null>
    / ( <identifier list> )
```

```

        <parameter declaration list>
<parameter declaration list> ::= <parameter declaration>
                           /<parameter declaration list>
                           <parameter declaration>
<parameter declaration> ::= <identifier list> :
                           <parameter type> <type>
                           /<identifier list> :
                           PROCEDURE
<parameter type> ::= VALUE
                   / NAME
                   / REFERENCE
                   / FUNCTION

```

The parameter names, according to the syntax structure specified above, are declared twice in the procedure (or function), the first to define the order that the values will be accessed at invocation, the second to define the parameter type and the declaration type of each name. The structure is similar to ALGOL except that all identifiers must be given a parameter type. (ALGOL defines the parameter type to be NAME by default.)

For value parameters, a value cell is created at invocation of the procedure and the value passed into the procedure is stored there. This provides protection against changing the value of a variable argument.

For name parameters, the argument is defined to be an expression. The "formula" of the expression is passed into the procedure and the interpreter delays evaluating the expression until it is needed.

For reference parameters, the address of the argument is passed into the procedure. The address is then used to reference the value needed by the procedure. This parameter type is used to return

values through the argument list. Figure 17 illustrates the three different types of parameters used in a procedure. The value of the expression C is determined when the array variable B is declared.

```
DECLARE
  .
  .
  .
  PROCEDURE examp_proc (a,b,c,d)
    a,c:NAME INTEGER
    b:VALUE ARRAY [c] OF REAL
    d:REFERENCE BOOLEAN
    DECLARE . . . END DECLARE
  .
  .
  .
  END PROCEDURE examp_proc
END DECLARE
```

FIGURE 17 Procedure Declaration Example

To provide uniformity with the semantics of ALGOL, a procedure or function should always be executed within the block environment in which it is declared (known as static binding). Like variables, once the subtask (or block) in which the procedure was declared has been exited, the procedure is no longer available unless the name of the procedure is passed as an argument outside of the block.

To extend the graphics capabilities of the language, special procedures could be syntactically defined by the system or by the user to generate a high order of two and three-dimensional primitive objects. These procedures are similar to the procedures described above, except that they are invoked with the DRAW command within a page or page segment subtask.

This version of ASGOL provides only the two-dimensional primitives LINE and ARROW. These could be easily extended to include such primitives as circles, ellipses, squares, rectangles, arcs, and crosses. Unit value parameters of these primitives should define the location and size of each type. User defined 2-D primitives could then be graphically generated using system primitives or other user 2-D primitive procedures. A 2-D primitive could be syntactically defined within the procedure declaration list as follows.

```
<procedure declaration list> ::=  
    <procedure declaration list>  
        <2-D declaration>  
    <2-D declaration> ::= <2-d head> <2-D body>  
        <2-D end>  
    <2-D head> ::= 2-D OBJECT <identifier>  
        <parameter list>  
    <2-D body> ::= <declaration list>  
        <segment structure command list>  
        <draw instruction list>  
    <2-D end> ::= END OBJECT <identifier>
```

The body of a 2-D object is restricted to include only the declaration of constants and variables, segment structure commands, and a sequence of draw instructions.

3-D primitives may be declared in a similar fashion.

```
<procedure declaration list> ::=  
    <procedure declaration list>  
        <3-D declaration>  
    <3-D declaration> ::= <3-D head> <3-D body>  
        <3-D end>  
    <3-D head> ::= 3-D OBJECT <identifier>  
        <parameter list>  
    <3-D body> ::= <declaration list>  
        <segment structure command list>  
        <draw instruction list>
```

```
<3-D end> ::= END OBJECT <identifier>
```

Unlike 2-D objects however, 3-D objects can have transformation parameters implicitly allocated at the object's declaration. The functions to perform the transformations could be built into the system. These parameters define the rotation (alpha, beta, and gamma), the translation, and the scaling of the object when the procedure is invoked. The rotation parameters define the rotation of the object in the X, Y, and Z plane respectively.

BIBLIOGRAPHY

1. Goodman, S. E. and S. T. Heintzeli. Introduction to the Design and Analysis of Algorithms. New York: McGraw-Hill Books, 1977.
2. Bohm and Jacopini. Algorithm Structures. New York: McGraw-Hill Books, 1969.
3. Gries, David. Compiler Construction for Digital Computers. New York: John Wiley & Sons, Inc., 1971.
4. ISSCO. DISSPLA User's Manual, Version 7.3. San Diego: Integrated Software Systems Corporation, 1970.
5. ISSCO. DISSPLA User's Manual, Version 8.1. San Diego: Integrated Software Systems Corporation, 1975.
6. ISSCO. DISSPLA User's Manual, Pocket Manual. San Diego: Integrated Software Systems Corporation, 1975.
7. Wetherell, Charles and Alfred Shannon. LR Automatic Parser Generator and LR(1) Parser. Livermore, Cal.: Lawrence Livermore Laboratory, June 1979.
8. Wirt, Michael C. Lecture Notes. Advanced Compiler Theory Course.
9. Jones, Ben. "An Extended ALGOL-60 for Shaded Computer Graphics," SIGGRAPH/SIGPLAY Proceedings on Graphics Languages: p. 18 (16-27 April 1976).
10. Amburn, Elton P. The Graphical Display of Multi-Dimensional Aerodynamic Flow Field Data. A language and system, called CINTMA, that uses the LR(1) parsing technique. Thesis, AFIT/GCS/MA/79D-1, Dec 1979.
11. Dahl, David A. MAPPER. A Graphics Oriented Language. Los Alamos, California: Los Alamos Scientific Laboratory, June 1972.

APPENDIX A
LR(1) BNF CODE OF ASGOL

```
<program> ::= <program head> <program body> <program end> .

<program head> ::=  program  <program identifier>

<program identifier> ::= <identifier>

<program body> ::= <declaration list> <section block list>

<program end> ::=  end program <program identifier>

<declaration list> ::= <null>

          / declare <constant declaration>

          end declare

<constant declaration part> ::= constant
                                <constant declaration list>

<constant declaration list> ::= <constant declaration>

          / <constant declaration list>

          <constant declaration>

<constant declaration> ::= <identifier> = <expression>

<variable declaration part> ::=

          / variable

          <variable declaration list>

<variable declaration list> ::= <variable declaration>

          / <variable declaration list>

          <variable declaration>
```

```
<variable declaration> ::= <identifier list> : <type>

<identifier list> ::= <identifier>
                    / <identifier list> , <identifier>

<type> ::= <basic type>
          / <array type>
          / string ( <string size> )
          / axis
          / unit

<string size> ::= <expression>

- <array type> ::= array <bounds> of <basic type>

<bounds> ::= <array bounds> ]

<array bounds> ::= [ <expression>
                     / <array bounds> , <expression>

<basic type> ::= integer
                  / real
                  / boolean
                  / character

<section block list> ::= <section structure command list>
                           <section listing>
                           / <section structure command list>
                           <page listing>
                           / <section structure command list>
```

```
        <segment block list>

<section structure command list> ::= <null>
        / <section structure command list>

                <section structure command>

<section structure command> ::= <assignment instruction>
        / <change instruction>
        / <page margin instruction>
        / <char height instruction>
        / <if instruction type1>
        / <while instruction type1>
        / <for instruction type1>
        / <repeat instruction type1>
        / <case instruction type1>

<section listing> ::= <section>
        / <section listing> <section>

<section> ::= <section head> <section body> <section end>

<section head> ::= section <section identifier>

<section identifier> ::= <identifier>

<section body> ::= <declaration list> <section block list>

<section end> ::= end section <section identifier>

<page margin instruction> ::= binding = <unit value>
        / grace = <unit value>
        / border = <unit value> by
```

```
        <unit value>

<page listing> ::= <page>
                  / <page listing> <page>

<page> ::= <page head> <page body> <page end>

<page head> ::= page <page identifier> ( <page parameter> )

<page identifier> ::= <identifier>

<page body> ::= <declaration list> <page block list>

<page end> ::= end page <page identifier>

<page parameter> ::= <direction> , <margin set> ,
                     <number> , <location>

<location> ::= <location signal> top
                  / <location signal> bottom

<location signal> ::= left
                     / right
                     / inside
                     / outside
                     / <null>

<number> ::= <expression>

<direction> ::= horizontal
                  / vertical

<margin set> ::= left reset
                  / right reset
```

```
    / center
    / <null>

<page block list> ::= <page structure command list>
                    <instruction>
    / <page structure command list>
                    <segment listing>

<page structure command list> ::= <null>
                    / <page structure command list>
                    <page structure command>

<page structure command> ::= <margin instruction>
                    / <frame instruction>
                    / <assignment instruction>
                    / <change instruction>
                    / <char height instruction>
                    / <if instruction type2>
                    / <while instruction type2>
                    / <for instruction type2>
                    / <repeat instruction type2>
                    / <case instruction type2>

<segment listing> ::= <segment>
                    / <segment listing> <segment>

<segment> ::= <segment head> <segment body> <segment end>

<segment head> ::= segment <segment identifier>
                    ( <segment parameter> )
```

```
<segment parameter> ::= <unit value> , <unit value> ,  
                           <unit value> , <unit value>  
  
<segment identifier> ::= <identifier>  
  
<segment body> ::= <declaration list> <segment block list>  
  
<segment block list> ::= <segment structure command list>  
                           <instruction>  
  
<segment structure command list> ::= <null>  
                           / <segment structure command list>  
                           <segment structure command>  
  
<segment structure command> ::= <margin instruction>  
                           / <frame instruction>  
                           / <assignment instruction>  
                           / <change instruction>  
                           / <char height instruction>  
                           / <if instruction type3>  
                           / <while instruction type3>  
                           / <for instruction type3>  
                           / <repeat instruction type3>  
                           / <case instruction type3>  
  
<segment end> ::= end segment <segment identifier>  
  
<margin instruction> ::= margin <margin values>  
  
<margin values> ::= ( <unit value> , <unit value> )  
  
<unit value> ::= <expression> <units>
```

```
<units> ::= inch
        / inches
        / <unit identifier>

<axis definition> ::= ( <title> , <type axis> , <min> ,
                         <max> , <delta> , <ticks> )

<title> ::= <character string>
        / <string identifier>

<type axis> ::= linear
        / log
        / logarithmic
        / month

<min> ::= <expression>
        / <month>

<max> ::= <expression>
        / <month>

<month> ::= jan
        / feb
        / mar
        / apr
        / may
        / jun
        / jul
        / aug
        / sep
```

```
    / oct
    / nov
    / dec

<delta> ::= <expression>

<ticks> ::= <expression>

<frame instruction> ::= frame <frame thickness>

<frame thickness> ::= ( <expression> )

<assignment instruction> ::= <set instruction>
    / <input instruction>
    / <output instruction>

<set instruction> ::= set <variable> = <expression>
    / set string <section identifier> =
        <expression>
    / set unit <unit identifier> =
        <unit value>
    / set axis <axis identifier>
        <axis specification>

<axis identifier> ::= <identifier>

<unit identifier> ::= <identifier>

<section identifier> ::= <identifier>

<axis specification> ::= = <axis definition>
    / . title = <title>
```

```
    / . type = <type axis>
    / . min = <min>
    / . max = <max>
    / . delta = <delta>
    / . ticks = <ticks>

<if instruction type1> ::= <if head> <true branch type1>
                           <false branch type1> end if

<if instruction type2> ::= <if head> <true branch type2>
                           <false branch type2> end if

<if instruction type3> ::= <if head> <true branch type3>
                           <false branch type3> end if

<if head> ::= if <expression>

<true branch type1> ::= then <declaration list>
                           <section block list>

<true branch type2> ::= then <declaration list>
                           <page block list>

<true branch type3> ::= then <declaration list>

<false branch type1> ::= else <declaration list>
                           <section block list>

<false branch type2> ::= else <declaration list>
                           <page block list>

<false branch type3> ::= else <declaration list>
```

```
<segment block list>

<while instruction type1> ::= <while head> do
    <section block list>
    end while

<while instruction type2> ::= <while head> do
    <page block list>
    end while

<while instruction type3> ::= <while head> do
    <segment block list>
    end while

<while head> ::= while <expression>

<for instruction type1> ::= <for head> <section block list>
    end for

<for instruction type2> ::= <for head> <page block list>
    end for

<for instruction type3> ::= <for head> <segment block list>
    end for

<for head> ::= for <variable> = <expression> to
    <expression> <by clause> do
/ for <variable> = <expression> down
    <expression> <by clause> do

<by clause> ::= by <expression>
```

```

<repeat instruction type1> ::= <repeat head>
                                <section block list>
                                end repeat

<repeat instruction type2> ::= <repeat head>
                                <page block list>
                                end repeat

<repeat instruction type3> ::= <repeat head>
                                <segment block list>
                                end repeat

<repeat head> ::= repeat until <expression>

<case instruction type1> ::= <case head> <case seq type1>
                                end case

<case instruction type2> ::= <case head> <case seq type2>
                                end case

<case instruction type3> ::= <case head> <case seq type3>
                                end case

<case head> ::= case <variable> of

<case seq type1> ::= <case list> <declaration list>
                                <section block list>
                                / <case seq type1>
                                <case list> : <declaration list>
                                <section block list>

<case seq type2> ::= <case list> : <declaration list>

```

```
<page block list>
/ <case seq type2>
<case list> : <declaration list>
<page block list>

<case seq type3> ::= <case list> : <declaration list>
<segment block list>
/ <case seq type3>
<case list> : <declaration list>
<segment block list>

<case list> ::= <int const list>
/ <character string list>
/ others

<character string list> ::= <character string>
/ <character string list> ,
<character string>

<variable> ::= <identifier> <array specification>

<array specification> ::= <bounds>

<input instruction> ::= input <identifier> : <source>

<source> ::= terminal
/ tape <int const>
/ <direct input>

<direct input> ::= data / <constant set> /
<constant set> ::= <int const set>
```

```

        / <real const set>
        / <boolean const set>
        / <character string>

<int const set> ::= <int const> <list number>
        / <int const set> ,
        <int const> <list number>

<real const set> ::= <real const> <list number>
        / <real const set> ,
        <real const> <list number>

<boolean const set> ::= <boolean value> <list number>
        / <boolean const set> ,
        <boolean value> <list number>

<list number> ::= : <int const>

<output instruction> ::= output <identifier> : <port>

<port> ::= terminal
        / tape <int const>

<change instruction> ::= change <from set> to <to set>

<from set> ::= <case set> roman
        / <case set> italic
        / <case set> script

<to set> ::= special
        / math
        / instruction

```

```

/ <case set> greek
/ <case set> russian
/ hebrew

<case set> ::= upper
/ lower

<character height instruction> ::= height = <unit value>

<expression> ::= <condition expression>
/ <condition expression> <logic operator>
<condition expression>

<condition expression> ::= <simple expression>
/ <condition expression>
<condition operator>
<simple expression>

<simple expression> ::= <term>
/ + <term>
/ - <term>
/ <simple expression> + <term>
/ <simple expression> - <term>

<term> ::= <factor>
/ <term> * <factor>
/ <term> / <factor>
/ <term> mod <factor>
/ <term> rem <factor>

<factor> ::= <primary>

```

```
    / <factor> ** <primary>

<primary> ::= not <primary>
    / ( <expression> )
    / <variable>
    / <constant>
    / <character string>
    / float ( <expression> )
    / integer ( <expression> )
    / fraction ( <expression> )
    / sin ( <expression> )
    / cos ( <expression> )
    / tan ( <expression> )
    / inv sin ( <expression> )
    / inv cos ( <expression> )
    / inv tan ( <expression> )
    / absolute ( <expression> )
    / point ( <section identifier> )
    / origin ( <x or y> )
    / area ( <x or y> )
    / reference ( <section identifier> ,
                  <character string> )
    / string ( <expression> )
    / length ( <section identifier> )

<x or y> ::= x
    / y

<logic operator> ::= <
```

```
/ <=
/ =
/ /= 
/ >=
/ >

<condition operator> ::= and
/ or
/ xor

<constant> ::= <int const>
/ <real const>
/ <boolean value>

<int const list> ::= <int const>
/ <int const list> , <int const>

<boolean value> ::= true
/ false

<instructions> ::= <draw instructure list>
/ <graph instruction>
/ <text instruction>

<draw instruction list> ::= <draw instruction>
/ <draw instruction list>
<draw instruction>

<draw instruction> ::= draw arrow <arrow style>
( <unit value> , <unit value>
, <unit value> , <unit value> )
```

```
 / draw line ( <unit value> , <unit value>
               , <unit value> , <unit value> )

<arrow style> ::= <int const>
               / <variable>

<graph instruction> ::= <graph preparation instruction list>
                         graph <title option>
                         ( stack form <frame option>
                           <interpolation type> ,
                           <number plots> , <plotting points> )

<graph preparation instruction list> ::= <null>
               / <graph preparation instruction list>
               <graph preparation instruction>

<graph prep instruction> ::= <graph id instruction>
               / <legend instruction>
               / <axis instruction>

<graph id instruction> ::= grid <title option>
                           ( <expression> , <expression> )

<legend instruction> ::= legend ( <location option>
                           <title list> )

<location option> ::= <location> ,
                     <location> , <location>

<title list> ::= <title>
               / <title list> , <title>

<axis instruction> ::= <axis> = <axis definition>
```

```
    / <axis> = <axis identifier>

<axis> ::= <x or y> axis

<title option> ::= <title>

<stack form> ::= stack of <int const>

<frame option> ::= framed

<interpolation type> ::= linear
    / step
    / bar
    / stacked bar
    / spline
    / smooth
    / pie

<number plots> ::= <expression>

<plot points> ::= <identifier>
    / <direct input>

<text instruction> ::= text ( <justification> <text style>
    , <section name> , <start> ,
    <length> , <size> )

<text style> ::= simple
    / cartog
    / complx
    / duplx
    / gothic
```

```
    / scmplx
    / simplx
    / triplx

<string name> ::= <string identifier>
    / <character string>

<start> ::= <expression>
    / next

<length> ::= <expression>
    / continue

<size> ::= <unit value>

<justification> ::= <horizontal format> justified
    / <vertical format> centered

<horizontal format> ::= left
    / right
    / l-r

<vertical format> ::= top
    / bottom
    / <null>

<null> ::=
```

APPENDIX B SYSTEM COMMANDS

There are four general commands that the system presently supports. These are CREATE, RUN, LIST, and END. All of the system commands were developed with the intention that later versions of this system would support interactive capabilities.

The CREATE command is used to transfer a program from the terminal (or input file) to the program input file. This provides an alternative to attaching a file containing the program text. A file TAPE7 is created with the CREATE command. Lines are read from the terminal until a "\$" is detected in the first column of the input line. An end-of-file marker is then written to the program input file and a ready-prompt "COMMAND=" is issued.

The RUN command tells the system to begin parsing the program stored in TAPE7. The subtask name following the RUN command specifies the block of graphics to be generated in the program. If the subtask name is the program name, the graphics for the complete program is generated; for section and page subtasks (defined in appendix C under Beginner's Guide), the graphics for that specific task is generated.

For page segment subtasks, the graphics instructions are drawn within the complete domain of the page. Frame and margin instructions defined within the page, but not within the segment, are ignored.

The LIST command copies all or any part of the program input file (TAPE7) or the program output file (TAPE10) to the terminal. To list the complete file, the commands are:

LIST, INPUT, ALL
LIST, OUTPUT, ALL

and to list specific lines of the file:

LIST, INPUT, <# lines>, <starting line>
LIST, OUTPUT, <# lines>, <starting line>

The END command stops execution of the program.

APPENDIX C PROGRAMMING GUIDE

One of the most difficult tasks in designing a new high-level language is not in the design of the instruction set that is available in the language, nor is it in the implementation of an interpreter or compiler to generate correct results from any grammatically correct program, but it is the description of the language in English terms. This is not to say that the design and implementation of a language are not important. But, without a complete description of the language for a user to follow, he or she cannot be expected to write grammatically correct programs in an efficient manner using the complete utilities available.

However, if the language is designed to allow the maximum amount of freedom in structuring and if the instruction set available is minimized to accomplish the desired results, as was the design purpose of this language, this task is simplified.

This appendix is divided into three sections: the beginner's guide, program enhancements, and string manipulation. The beginner's guide gives an introduction to program structuring and common instructions available in the language for generating simple graphs and text. The section on program enhancement describes the use of constants and variables, the instruction set used for declaring and defining variables, and the structure of arithmetic evaluations. The third section discusses the instructions that define the various fonts and styles for characters drawn to the screen.

i. Beginner's guide

Before discussing the structure of a program for this graphics

language, several terms used in this appendix need to be defined. The definitions of these terms should not be considered universal in scope, since several of the terms pertain specifically to this language.

ALGORITHM: An unambiguous step by step sequence of instructions used to describe how to perform a task or procedure. A food recipe is an example of an algorithm.

FLOWCHART: A directed network of instruction blocks completely connected by unidirectional paths between the blocks.

TOP-DOWN BLOCK STRUCTURING: A step by step refinement of an algorithm into successively smaller tasks or subtasks.

TASK: A definition of a desired result.

PROGRAM: Any of various methods used to define the order in which tasks are to be accomplished. A program varies from an algorithm in that the program is written in the format of a specific language.

PAGE SUBTASK: The set of instructions used to generate a single page of graphics.

SUBPLOT AREA: The rectangular area in which all text, lines, and graphs are to be drawn for the page or page segment.

PHYSICAL ORIGIN: The lower left corner of the subplot area defined for the page or page segment.

LANGUAGE INSTRUCTION: A sequence of words and symbols grammatically specified by the language and used to perform a specific part of the subtask in which it is embedded.

PARAMETER: A value used by an instruction to perform an operation.

RESERVED WORD: Any of the 166 English words and legal symbols used for special significance in the language. A complete list of these words and symbols are attached at the end of this appendix for the user.

IDENTIFIER: A sequence of letters and digits not separated by blanks, declared by the user for a single specific purpose. There are four restrictions upon the use of identifiers:

- 1) an identifier must begin with a letter (A through Z);
- 2) only letters, digits (0 through 9), and the underscore (_) are allowed in legal identifiers;
- 3) the length of the identifier must not exceed 10 characters; and
- 4) an identifier cannot be a reserved word.

Other than these restrictions, a user has complete freedom in defining names for identifiers.

LITERALS: There are four classes of literals: integer literals, real literals, character literals, and boolean literals.

INTEGER LITERAL: The character representation of a finite number of consecutive digits. Examples are: 1026 , 0 , -81 , 1000 .

REAL LITERAL: The character representation of an integer number followed by a period (.) And a fractional value. Examples are: 3.14159 , -16.103 , .10 , 1. , 100000. .

CHARACTER LITERAL: A single letter, digit, or symbol enclosed in double quotations. Examples are: "g","(","5","." .

CHARACTER STRING: Any finite sequence of characters (letters, digits, or symbols) enclosed in double quotations (""). Examples are:

"THIS IS A STRING" produces the string:

THIS IS A STRING

"" produces the empty string of length 0

"""" produces a single double quotation "

It is important to note that any use of the double quotation within a string must be printed twice for each double quote desired.

"""""" produces a double double quotation ""

"" produces a single quotation '

"The & symbol is used to continue"&

"a string to the next line."

In drawing characters on the screen, six special characters are used to define the font to be used. These characters are (,), +, -, *, and /. To print any of these characters, enclose them in parentheses(i.e. To print a "(" on the screen, the character sequence would be "((())". In section III of this appendix, it will be shown how shift characters are used to determine fonts.

BOOLEAN LITERAL: the two literals TRUE and FALSE that define the result of a condition.

COMMENTS: Comments begin with the symbols /* anywhere within a program and end with the two symbols typed in reverse: */. They have no effect upon the execution of the program and are used for the sole purpose of program documentation. It helps the reader understand what the program is doing. Examples of comments are:

/* this is a comment */

```
/*      comments can be
        formatted in almost any style
        and can be stretched across any
        number of lines.          */
```

A special caution should be made to insure that all comments end with the */ symbols. Without these to end a comment, program instructions will be bypassed between comments.

The language provides a second method of comments. With the double character set --, all text to the end of the line is treated as comments.

Program Block Structuring:

The first step in writing a program for this graphics language (or for any other language) should be to define the subtasks of the program in a top-down structure. A subtask is a fragment of the program such that the sequential execution of all of the subtasks represent the complete program. Each subtask can be divided into smaller subtasks, if necessary, which again can be divided into even smaller subtasks, until a point is reached when each subtask represents the block of instructions used to generate graphics for a single page (called a page subtask). This block structuring or nesting of subtasks within larger tasks allows versatility in the program, discussed in the next section.

All programs in this language begin with the reserved word PROGRAM followed by a unique identifier to define the name of the program. The program halts on the recognition of the reserved words END PROGRAM followed by the name of the program and a period (.).

A program subtask (but not a page subtask) begins with the reserved word SECTION followed by an unique identifier to name the section. The section ends with the reserved words END SECTION followed by the name of the section. As can be seen in figure 18, section subtasks can be declared within sections.

```
PROGRAM prog_name
.
.
.
SECTION sec_name_1
.
.
.
END SECTION sec_name_1
SECTION sec_name_2
.
.
.
SECTION internal_1
.
.
.
END SECTION internal_1
SECTION internal_2
.
.
.
END SECTION internal_2
END SECTION sec_name_2
.
.
.
END PROGRAM prog_name
```

FIGURE 18 Program Block Structure

A page subtask begins with the word page, followed by a unique

identifier to define the name of the page, and a page construct list.

The page ends with the reserved words END PAGE followed by the name of the page.

The page construct list is used to define the physical construction of that page. It is a list of four parameters enclosed in parentheses and separated by commas. These parameters will be denoted as the page direction, the page margin, the page number, and the page number location respectively. (It is important to note at this point that the order of these parameters, as well as the parameters used in all other instructions, is important. Any variation in the order specified will produce errors.)

The first parameter, the page direction, determines whether the page format is to be HORIZONTAL or VERTICAL by placing one of these two reserved words as the parameter. If the format is horizontal, the page dimension will be 11 inches in the horizontal direction and 8 1/2 inches in the vertical direction. If the format is vertical, the dimensions are reversed: 11 inches in the vertical direction, 8 1/2 inches in the horizontal direction. To keep from getting confused, just remember that the longest side (11 inches) is always in the direction specified in the parameter.

The page margin defines how the grace margin of the page is to be set up. If the reserved words LEFT RESET are used as the parameter, the margins of the page will be set to be 1 inch on the upper, lower, and the right sides of the page, and 1 1/2 inches on the left side. (In the horizontal format, the upper and lower edges are the 11 inch sides of the page; in the vertical format, they are

the 8 1/2 inch sides.) The 1 1/2 inch margin along one side of the page is called the binding margin and allows space for binding the pages together into book form. If the reserved words RIGHT RESET are used, the grace margin will be set to 1 inch along the upper, lower, and left sides of the page, and 1 1/2 inches along the right. This resets the binding margin to the right side of the page. If the reserved word CENTER is used as the parameter, the subplot area will be centered on the page with 1 inch margins on all four sides.

The page number parameter can be either an integer value, a real value, or a character string. If the integer value 0 is used, no page number will be printed. In the case of a character string, a maximum of twenty characters are allowed. If more than twenty characters are used, a warning will be issued, and only the first twenty will be printed.

The last parameter, the page location, specifies any of six positions on the page the page number is to be printed. If the reserved word TOP is used as the parameter, the page number will be centered and printed at the top of the page, 1/2 inch from the edge. If TOP is preceded by the reserved word LEFT, the number will be printed at the left top edge, beginning directly above the left margin of the page (1 1/2 inches from the left edge for left binding margins, 1 inch from the left edge for right binding margins or centered). If TOP is preceded by the reserved word RIGHT, the number will be printed at the right top edge, ending directly above the right margin.

For pages that are not CENTERed (that have a binding margin),

AD-A100 814 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOO--ETC F/6 9/2
ASGOL--AN ALGOL-STRUCTURED GRAPHICS ORIENTED LANGUAGE. (U)
MAR 81 J D HART

UNCLASSIFIED AFIT/GCS/MA/81M-2

NL

2 OF 2
AD A
100814

END
DATE
FILED
7-31
DTIC

TOP may be preceded by one of two other reserved words: INSIDE or OUTSIDE. If INSIDE TOP is used, the page number will be printed on the inside of the page, either beginning or ending directly above the binding margin, depending on whether the page has a left binding margin or a right binding margin respectively. If OUTSIDE TOP is used, the number will be printed directly opposite of the binding margin.

If the reserved word BOTTOM is used as the page location parameter, the page number will be centered at the bottom of the page, 1/2 inch above the lower edge. The reserved words LEFT, RIGHT, INSIDE, and OUTSIDE also can be used in the same manner as in TOP.

Figure 19 illustrates two ways pages can be defined. In the first page, VERT_SET, the page format is vertical, the subplot area is centered on the page, and the page is not numbered. In HORIZ_SET, the page format is horizontal, a binding margin is defined on the left side of the page, and A-5 is printed at the top center of the page.

```
.
.
.
SECTION example
.
.
.
PAGE vert_set(VERTICAL,center,0,right bottom)
.
.
.
END PAGE vert_set
PAGE horiz_set(HORIZONTAL,left reset,"a-5",top)
.
.
.
END PAGE horiz_set
END SECTION example
.
.
.
```

FIGURE 19 Page Examples

Often, it is necessary to draw several graphs, or to mix graphs and text, on a single page. To give this capability, the page subtask can be subdivided into page segment subtasks. Each page segment subtask begins with the reserved word segment, followed by a unique identifier (to define the name of the segment) and a segment parameter list. The segment ends with the reserved words END SEGMENT, and just as in section and page subtasks, it must be followed by the name of the segment.

The segment parameter list is a list of four integer or real values, each followed by the reserved word INCH or INCHES. Each parameter is separated by a comma and the complete parameter list is

enclosed in parentheses. These four parameters specify the size and location of the segment within the physical page subplot in which the subtask is to be performed.

The first parameter is the distance in the X direction from the physical page origin to the start of the segment subplot area. (From this point on, the X direction will refer to the dimension of the page or segment along the horizontal and the Y direction along the vertical dimension of the page or segment.) The second parameter is the distance in the X direction from the physical page origin to the end of the segment subplot area.

The third parameter is the distance in the Y direction from the physical page origin to the start of the segment subplot area. And the fourth parameter is the distance in the Y direction to the end of the segment subplot area.

These distances are specified by either integer or real values and are in inches. Several precautions should be considered in defining the segment subplot area. First, the values of the second and fourth parameters must be greater than the values of the first and third parameters respectively. This should seem reasonable since the end of the subplot area cannot be defined to be before the beginning. Second, none of the four parameters values can lie outside of the physical page, and for this reason, none of the four parameters can have negative values.

As an example of the structuring of a program as discussed so far, consider a single page task program that creates two graphs drawn side by side in the upper half of the page, leaving the lower

half is to be reserved for a text description of the graphs. The block structuring of this program is shown in figure 20. The reader should study this structuring before continuing on to the next subsection.

```
PROGRAM example_1
.
.
.
PAGE sampledraw(VERTICAL,CENTER,"sample",TOP)
.
.
.
SEGMENT graph1(0 INCHES,4.25 INCHES,
                5.5 INCHES,11 INCHES)
.
.
.
GRAPH
END SEGMENT graph1
SEGMENT graph2(4.25 INCHES,8.5 INCHES,
                5.5 INCHES,11 INCHES)
.
.
.
GRAPH
END SEGMENT graph2
SEGMENT texter(0 INCHES,8.5 INCHES,0 INCHES,5.5 INCHES)
.
.
.
TEXT
END SEGMENT texter
END PAGE sampledraw
END PROGRAM example_1 .
```

FIGURE 20 Segment Examples

In the above program listing, EXAMPLE_1 is the name of the program, SAMPLEDRAW is the name of the page, and GRAPH1, GRAPH2, and TEXTER are segment names. The page format is vertical with a 1 inch margin on all sides. The string SAMPLE is the page number and

is printed at the center top of the page. The GRAPH1 subplot area is the upper left quarter of the page; GRAPH2's subplot area is the upper right quarter of the page; and TEXTER 's subplot area is the lower half of the page.

The reserved words GRAPH and TEXT are used in this example to represent the points at which these instructions occur. The actual use of these instructions will be discussed later in this section.

Structure Commands:

In this section, two structure commands will be presented. These instructions are only allowed within a page or page segment subtask .

- The margin instruction begins with the reserved word MARGIN, and may be followed by two parameter values separated by a comma and enclosed in parentheses. These parameter values are integer or real values followed by the reserved word INCH or INCHES. These parameters reset the subplot area by defining the size of the margin for both the X and the Y domains respectively. Figure 21 shows an example of this instruction. In this figure, the margin instruction redefines the physical origin, which was previous to the margin instruction, at 1 inch in the X direction and 1 inch in the Y direction, to be 2.5 inches and 3 inches respectively. These values were obtained by adding the margin values (1.5,2) to the existing physical origin (1,1). The margin instruction also redefines the subplot area, which was 6 inches vertically and 9 inches horizontally prior to the margin instruction (remember that VERTICAL defines the page size to be (8.5,11) and RIGHT RESET uses a 1.5 inch binding

```
margin and a 1 inch grace margin), to be 3 inches (6 - (1.5 x 2)) and  
5 inches (9 - (2 x 2)).
```

```
•  
•  
•  
PAGE example 2(VERTICAL,RIGHT RESET, . . .  
MARGIN(1.5 INCH,2 INCH)
```

```
•  
•  
•
```

FIGURE 21 Margin With Parameters

If the parameters are not specified (i.e. Only the reserved word MARGIN is used), a default value of 5% of the subplot area dimension is used for both directions. In figure 22, the margin instruction will produce default values of .3 inches (6 times .05) in the X direction and .45 inches (9 times .05) in the Y direction.

```
•  
•  
•  
PAGE example_2(VERTICAL,RIGHT RESET, . . .  
MARGIN
```

```
•  
•  
•
```

FIGURE 22 Margin Without Parameters

Special precautions should be made to insure that the margin instruction does not redefine one or both of the subplot dimensions to be less than zero. If so, errors will be generated. A simple

arithmetic check can be made to insure this does not happen. The X dimension must be greater than twice the X value prior to the instruction and the Y dimension must be greater than twice the Y value. Since the default values use a percentage of the subplot area, errors should not result unless too many margin instructions are made within a subtask.

The margin instruction also resets the physical origin of the subtask. Although figure 23 may look harmless, it will produce errors since after the margin instruction, the subplot area dimensions are 3 inches ($6 - (2 \times 1.5)$) and 5 inches ($9 - (2 \times 2)$) and the second and fourth parameters of the segment block are out of the page bounds!

```
•  
•  
•  
PAGE example_3(VERTICAL,RIGHT RESET, . . .  
  MARGIN(1.5 INCH,2 INCH)  
  SEGMENT(0 INCH,3.5 INCH,2 INCH,6 INCH)  
•  
•  
•
```

FIGURE 23 Margin Error Example

The frame instruction is used to draw a physical frame around the defined subplot area of the subtask. It begins with the reserved word FRAME and may be followed by an integer value enclosed in parentheses. This integer value represents the thickness of the frame to be drawn. If the parameter is not specified (i.e. Only the reserved word FRAME is used), a default value of a single line is used. The integer value in the parameter cannot exceed 25. If a

thickness greater than 25 lines is desired, consecutive frame instructions can be used.

Like the margin instruction, the frame instruction affects the physical origin and subplot area of the subtask, except on a much smaller scale. The physical origin is redefined by 1/100th of an inch for every line drawn, and consequently, the subplot area is redefined to be 2/100th of an inch less in both dimensions for every line drawn. For example,

FRAME(25)

will produce a frame 1/4 of an inch (25 times 1/100) thick; the physical origin will be redefined as having .25 inches added to both directions, and the subplot area will be .5 inches less in both dimension lengths.

Graphics Instructions:

Once a page or page segment subtask has been defined and the margin and frame, if desired, are specified, the system is ready to draw the graphics for the subtask. There are three basic types of graphics that are available: lines and arrows, graphs, and text. For each subtask, only one of these three can be used. In other words, these basic types cannot be mixed within a subtask. However, if a combination of these types within a page or segment area is desired, the area can be defined in several page segment subtasks. For example, if text is to be drawn along with a graph on a page called EXAMPLE_3, the format may be as shown in figure 24.

```
•  
•  
•  
PAGE example_3(VERTICAL,LEFT RESET, . . .  
SEGMENT part_1(0 INCH,6 INCH,0 INCH,9 INCH)  
•  
•  
•  
graph(. . .  
END SEGMENT part_1  
SEGMENT part_2(0 INCH,6 INCH,0 INCH,9 INCH)  
•  
•  
•  
TEXT( . . .  
END SEGMENT part_2  
END PAGE example_3  
•  
•  
•
```

FIGURE 24 Text and Graph Combination

In this example, note that both page segment subtasks called PART_1 and PART_2 define the dimensions to be that of the full page. Also, no margin or frame was defined at the head of the page, since these can be defined within the segment subtasks.

Line and Arrow Instructions:

A sequence of straight lines can be drawn anywhere within a page or page segment subtask by a series of line instructions. A line instruction begins with the reserved words DRAW LINE followed by four parameters enclosed in parentheses. The instruction's first two parameters are the X and Y coordinates of the starting location of the line specified in inches from the physical origin of the subtask. The third and fourth parameters are the X and Y coordinates of the ending location of the line from the physical origin of the subtask. It is insignificant which end of the line is specified first, as long as both coordinate points lie within the range of the subtask's

dimension. If a point lies outside of the subplot area, the line is clipped at the subplot edge. Figure 25 shows the use of this instruction. In this example, the page is defined to be centered and horizontal, defining the dimensions of the page to be 9 inches by 6 1/2 inches in the X and Y direction respectively. The margin instruction then redefines the X direction to be the same as the Y (6.5 inches) by the formula ((9 - 2.5) divided by 2). Note that the Y dimension is not changed by passing 0 INCHES as the parameter. The line instructions are then used to draw a diamond shaped box within the subtask.

```
•  
•  
•  
PAGE example_4(HORIZONTAL,CENTER, . . .  
MARGIN (1.25 INCHES,0 INCHES)  
DRAW LINE (0 INCH, 3.25 INCH,  
           3.25 INCH, 0 INCH)  
DRAW LINE (3.25 INCH, 0 INCH,  
           6.5 INCH, 3.25 INCH)  
DRAW LINE (6.5 INCH, 3.25 INCH,  
           3.25 INCH, 6.5 INCH)  
DRAW LINE (3.25 INCH, 6.5 INCH,  
           0 INCH, 3.25 INCH)  
END PAGE example_4  
•  
•  
•
```

FIGURE 25 Line Instruction

Arrows are drawn in a similar fashion. In fact, the two can be mixed in any sequence. An arrow instruction begins with the reserved words DRAW ARROW, followed by an integer value representing the arrow style, and four parameters enclosed in parentheses. The four parameters are the coordinates of the starting and ending locations

of the arrow in the same form as in the LINE instruction, except that the direction the arrow is drawn is significant. Unless the direction of the arrow heads is specified to point in both directions (see the description of the integer value below), the arrow head is drawn to the end location of the arrow (the third and fourth parameter location).

The integer value in the instruction is a two digit integer used to describe the arrow style. The first, a digit between 0 and 3, describes the form of the arrow head. The second digit, also between 0 and 3, defines the location of the arrow head. Figure 26 shows the forms and locations of arrow heads. Note that if the second digit is 0, the arrow has no head and this reduces to a simple line with the same parameters.

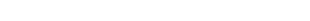
Forms:	0	solid head	
	1	white head	
	2	open head	
	3	closed head	
Locations:	0	none	
	1	end point	
	2	both points	
	3	both toward end	

FIGURE 26 Arrow Forms and Locations

Figure 27 gives an example of the use of arrows and lines combined. The page format and draw instructions are the same as those in figure 25. Arrow instructions have been added between line

instructions to show that the sequence order is unimportant. Figure 28 shows the result of this page subtask.

```
•  
•  
•  
PAGE example_4(HORIZONTAL,CENTER, . . .  
MARGIN(1.25 INCHES,0 INCH)  
DRAW LINE(0 INCH,3.25 INCH,3.25 INCH,0 INCH)  
DRAW LINE(3.25 INCH,0 INCH,6.5 INCH,3.25 INCH)  
DRAW ARROW 01(3.25 INCH,3.25 INCH,  
          0 INCH,6.5 INCH)  
DRAW ARROW 32(0 INCH, 6.5 INCH,  
          6.5 INCH,6.5 INCH)  
DRAW LINE(6.5 INCH,3.25 INCH,3.25 INCH,6.5 INCH)  
DRAW ARROW 03(0 INCH,0 INCH,3.25 INCH,6.5 INCH)  
DRAW LINE(3.25 INCH,6.5 INCH,0 INCH,3.25 INCH  
END PAGE example_4  
•  
•
```

FIGURE 27 Line and Arrow Instructions

page border

physical origin

after MARGIN

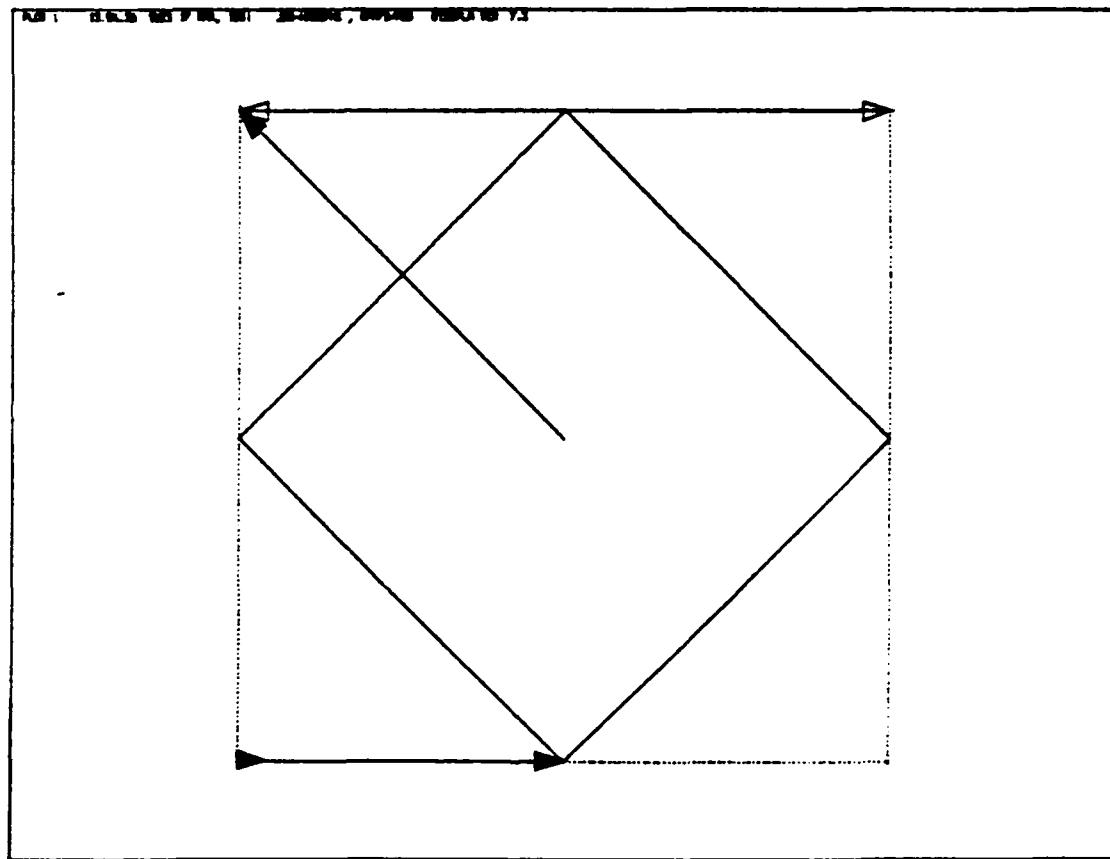


FIGURE 28 Lines and Arrows Example

Graph Instruction:

The graph instruction is a general purpose instruction used to draw any of a number of types of graphs. Only the basic types: linear, bar, and pie graphs will be discussed in this section. Other types of graphs will be discussed in the next section. At this point, the description of this instruction will be limited to these basic types.

Before discussing the graph instruction, there are three types of instructions, used to prepare the subplot area for a graph, that need to be described. These instructions must be placed immediately prior to the actual graph instruction. However, not all of these instructions are necessary to draw graphs but can be used to enhance the style of the graph.

The first type of instruction defines the X and Y axes of a graph. For linear and bar graphs, both axes must be defined. If these instructions are inserted prior to a pie graph, they will have no effect. The definition of an axis is given in five parts: the title of the axis, the type of axis, the minimum and maximum values, a step increment called delta, and the number of tick marks between each step.

The title of the axis is a user defined character string (remember a character string is enclosed in double quotations) of 80 characters or less. For the X axis, the character string is printed directly below the axis and centered within the subplot area. For the Y axis, the string is printed 90 degrees from the horizontal, directly to the left of the Y axis, and centered within the subplot area.

There are three types of axes available to the user: LINEAR, LOG or LOGARITHMIC, and MONTH. For a linear axis, the minimum and maximum values may be integer or real and represent the actual values of the end points of the axis. For a logarithmic axis, the minimum and maximum values are integers representing the exponent values of the end points (10 to the power of). For month axes, the minimum and maximum values are the first three letters of the month (i.e. MAY, JUN, JUL, and AUG) for the end points of the axis.

The delta is an integer value representing the number of incremental steps of the axis starting with 1 at the minimum value. When the axes are drawn in the subplot area, the values at each step will be drawn, as well as the minimum and maximum values of the axes. In defining the X axis of a linear or bar graph, the delta also represents the number of points to be plotted within the graph. Examples of the use of the delta are shown in figures 29 and 30.

```
    .  
    .  
    .  
X AXIS =("x axis1",MONTH,DEC,DEC, 5,2)  
Y AXIS =("y axis1",LINEAR, 0,100,6,0)  
GRAPH ( LINEAR, . . .  
    .  
    .  
    .
```

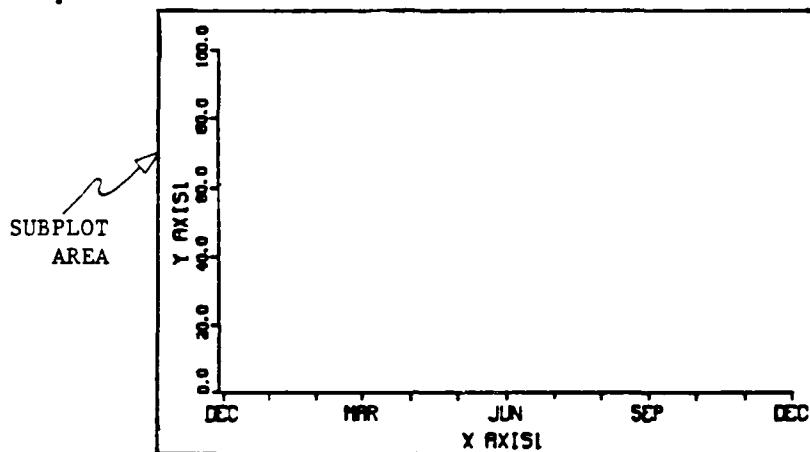


FIGURE 29 Linear Axes Definition

```
    .
    .
    .
    .
    Y AXIS =("log",LOG,-1,7,5,1)
    X AXIS =("linear",LINEAR, -100,100,3,0)
    GRAPH(BAR, . . .
```

```
    .
    .
    .
```

SUBPLOT
AREA

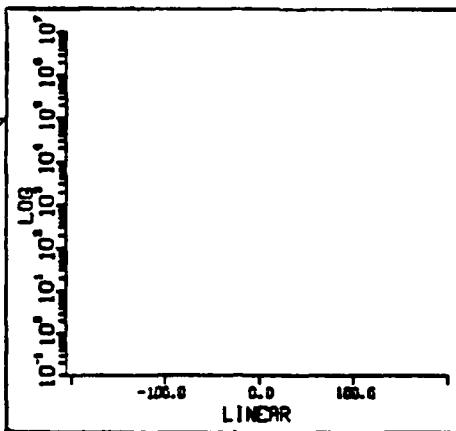


FIGURE 30 Bar Axes Definitions

The number of tick marks is an integer value that represents the number of marks placed evenly between each incremental step of the axis. Ticks do not effect the plotting of points on the graph, but are used for plotting enhancements. If tick marks are used, the values at the tick marks are not printed.

To define the X axis, the instruction is of the form:

```
X AXIS =(title,type,min,max,delta,ticks)
```

where title is the character string name of the axis; type is one of the reserved words: LINEAR, LOG, LOGARITHMIC, or MONTH; min and max are: integer or real values for LINEAR type axes, integer values for LOG or LOGARITHMIC type axes, or the first three letters of a month

for MONTH type axes; delta and ticks are integer values. The reserved words LOG is a short form of LOGARITHMIC.

To define the Y axis, the instruction is of the form:

Y AXIS =(title,type,min,max,delta,ticks)

where each of the parameters are the same as in the X axis definition above.

In figure 29, the instructions set up the axes system within the subplot area for a linear graph. In figure 30, the instructions set up the axes systems within the subplot area for a bar graph. For bar graphs, the minimum and maximum points of the X axis are spaced evenly within the axis to prevent bunching-up of the bars as they are being drawn at these points. Note also that it is not important which axis is defined first.

The second type of instruction, the legend instruction, is optional and may be used with any of the graph types. In a linear graph, a legend is used to identify the types of curve markers to each plot of the graph. In a bar graph, a legend identifies the shade styles to each bar plot drawn. In a pie graph, the legend is used to identify each of the pie sections to user defined character strings.

The legend instruction may be specified in any one of four various forms. These are:

LEGEND (title list)

LEGEND (location,title list)

LEGEND title (title list)

LEGEND title (location,title list)

The title list, used in all four variations, is the list of character strings (enclosed in double quotations), separated by commas. The length of these strings must be 20 characters or less. If the length is greater than 20, only the first 20 characters will be used. These character strings are the user defined strings identifying the plots on the graph. This sequence of character strings must be in the same order as the curves or bar plots being plotted on the graph.

The location parameter, used in the second and fourth variations, specifies the positioning of the legend box within the subplot area of a linear or bar graph. Any of the reserved word sets: LEFT TOP, TOP, RIGHT TOP, LEFT BOTTOM, BOTTOM, or RIGHT BOTTOM may be used as this parameter. If the location of the legend is not specified (in the first and third variations), the default is RIGHT BOTTOM. It is important to remember that points will not be plotted within the legend box if a legend is used; therefore, precautions must be made in positioning the legend box within the subplot area.

The title in the third and fourth variations after the reserved word LEGEND is a character string of 20 characters or less used to redefine the name of the legend. By default, "LEGEND" will be printed at the top of the legend box.

For a pie graph, if either the optional legend title or the location is used within the legend instruction, they will be ignored since the character strings are placed within the corresponding pie segments.

The third preparation instruction, the grid instruction, is also

optional and may be used with either the linear or bar graph. The grid instruction is used to draw solid or dotted lines across the subplot area along either the X axis, the Y axis, or both. The grid instruction begins with the reserved word GRID followed by two integer values enclosed in parentheses and separated by a comma. The two integer values represent the number of lines to be drawn per incremental step of the axis: the first for the X axis, the second for the Y axis. If the value is 0, no lines will be drawn. If the value is 1, a line is drawn at every step. If the values are positive, solid lines will be drawn. To draw dotted lines, the values must be negative (i.e. -2 will draw two dotted lines per incremental step of the axis).

Once the axes (for linear and bar graphs) and the optional legend and grid instructions have been defined, the subplot area is ready to plot the points on the graph. The graph instruction is specified in the form:

```
GRAPH title (graph type,number of plots,values)
```

The title is an optional user defined character string of length 20 characters or less. The title is printed at the top of the graph and centered across the subplot area. The graph type is one of the reserved words LINEAR, BAR, or PIE used to specify which basic type of graph is to be drawn. The other types of graphs will be discussed in the next section. The number of plots is an integer value that specifies the number of plots to be drawn on the linear or bar graph; for pie graphs, this value represents the number of pie sections in the graph.

Since the use of the values supplied in the graph instruction varies with the type of graph being drawn, this description is subdivided into two parts: pie graphs, and linear and bar graphs. At present, the description of this parameter is limited to direct input of the values. In the next section, it will be shown how these values can be passed into the instruction by an array identifier containing the values of the plotting points.

For direct input of the values, this parameter begins with the reserved word DATA followed by the list of values enclosed in slash marks ("/") and separated by commas. These values may be of type integer or real and may even be intermixed.

- Pie Graphs:

The values used in a pie graph are the percentages (%) of the pie sections. The sum of these percentages must be less than or equal to 100%. If the sum is less than 100%, the remaining section of the pie will have the character string "OTHERS" inserted. Figure 31 shows an example pie graph from a program, using the legend instruction. Figure 32 shows the results of this page subtask.

```

PROGRAM examplepie
  PAGE drawpie(VERTICAL,CENTER,5,RIGHT TOP)
  MARGIN(1 INCH,1 INCH)
  FRAME
  LEGEND ("cars",      /* automobiles */
          "trucks",     /* 2 and 4 wheel */
          "vans")      /* customized */
  GRAPH "automobile sales"
    (PIE,          /* define pie graph */
     3,            /* # of sections */
     DATA / 47.6, /* car sales */
     32.4,        /* truck sales */
     15/)         /* van sales */
  END PAGE drawpie
END PROGRAM examplepie.

```

FIGURE 31 Pie Graph Example

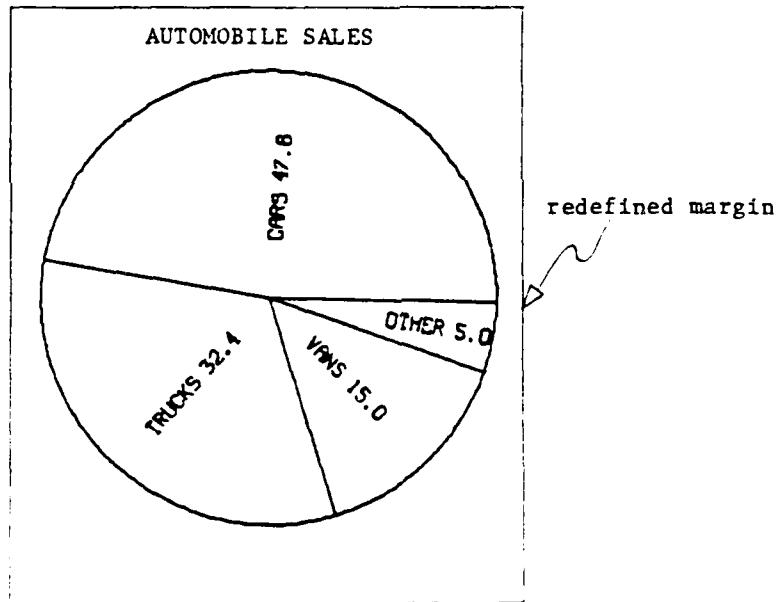


FIGURE 32 Pie Graph Example

Linear and Bar Graphs:

In linear and bar graphs, the values passed by the graph

instruction are the Y coordinates for each plotting point of the graph. Remember that the X coordinates are specified by the delta declared in the X axis of the graph. Therefore, the first value is the Y coordinate of the first plot; the X coordinate is at the minimum point of the X axis. The second value is the Y coordinate of the second point on the first plot; the X coordinate is at the first incremental step of the X axis. The sequence continues as shown in table 2.

In general, if the number of plots specified in the graph instruction is an integer value m , and if the delta of the X axis is specified as an integer value n , then the values defined in the graph instruction are the Y coordinates of the plots and the X coordinates are those shown in table 2.

TABLE 2

Direct Data Graph Coordinates	
VALUE NUMBER	X COORDINATES
1	minimum,1st plot
2	1st delta,1st plot
:	:
n	maximum,1st plot
$n+1$	minimum,2nd plot
:	:
$2n$	maximum,2nd plot
$2n+1$	minimum,3rd plot
:	:
mn	maximum,nth plot

As can be seen in table 2, the number of values used for plotting points is always m times n . If the number of values supplied in the graph instruction is less than this amount, the system will use the value of 0 for the remaining Y coordinates. If

the number of values is greater, then all values after the MNth will be ignored. Figure 33 and 34 show the use of the linear and bar graphs by plotting the same values for each. The value 36.8:5 represents the use of a feature available in direct input and has the same meaning as 36.8, 36.8, 36.8, 36.8, 36.8 .

```

PROGRAM exampleb_1
PAGE drawb_1(HORIZONTAL,LEFT RESET,0, TOP)
SEGMENT draw_bar(0 INCH, 4.25 INCH,      /* x dimen */
                  0 INCH, 6.5 INCH)      /* y dimen */
  FRAME /* enclose plot */
  /* scaling of axes within the subplot area
   will occur automatically */
  X AXIS =("hours",           /* hours of the day */
            0,24,             /* min and max */
            5,                /* # of plots */
            0)                /* no ticks */
  Y AXIS =("decibels",        /* noise level */
            0,120,            /* 0-120 Db */
            7,                /* every 20 Db */
            0)                /* no ticks */
  LEGEND "freeway"          /* title legend */
    (LEFT TOP,           /* location */
     "weekdays",
     "weekends",
     "average")          /* note order */
GRAPH(BAR,3,
      DATA/10,70,56,87,10, /* weekdays */
      6,37,43,40,9,      /* weekends */
      36.8:5)            /* average */
END SEGMENT draw_bar
SEGMENT draw_lin
  (4.25 INCH,8.5 INCH, /* x dimen */
   0 INCH,6.5 INCH)    /* y dimen */
X AXIS =("hours",0,24,5,5)
Y AXIS =("decibels",0,120,7,1)
GRAPH "noise level"(LINEAR,3,
      DATA /10,70,56,87,10,
      6,37,43,40,9,
      36.8:5 /)
END SEGMENT draw_lin
END PAGE drawb_1
END PROGRAM exampleb_1.

```

FIGURE 33 Bar and Linear Graph Example

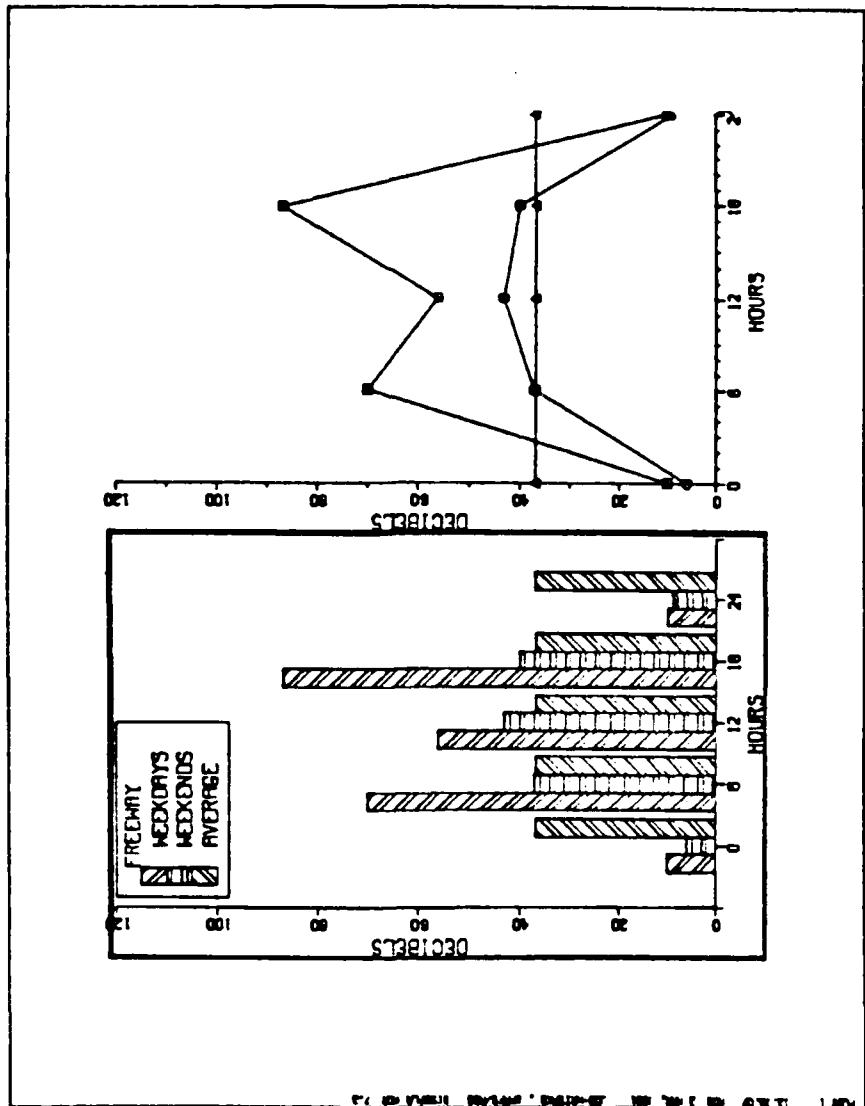


FIGURE 34 Bar and Linear Graph Example

Text Instruction:

The text instruction is used to print actual character strings in the page or segment subtask once the subplot area has been defined. All text instructions are of the form:

```
TEXT(text type text style,character string,  
      start,length,height)
```

The text type is the justification of the character string to be printed, determined by one of the reserved word sets: LEFT JUSTIFIED, RIGHT JUSTIFIED, L-R JUSTIFIED, TOP CENTERED, BOTTOM CENTERED, or CENTERED. With the reserved words LEFT JUSTIFIED, the character string will be printed in consecutive lines, starting at the top of the subplot area with left margin justification. The breaking point at the end of each line is determined by the number of words that can be packed per line. However, a string can be continued to the next line by inserting either "#nl" (new line) or "#np" (new paragraph) within the character string at the point the new line is to begin. "#nl" and "#np" are bypassed and will not be printed. "#nl" will cause the next line to be printed starting at the left edge of the subplot area "#np" will cause the next line to begin with a 15% indentation of the X dimension in the subplot area.

With the reserved words RIGHT JUSTIFIED, the character string will be printed with right margin justification. Breaking point of each line is determined in the same manner as in left margin justification, except that both "#nl" and "#np" are interpreted as "#nl". There is no paragraph indentation for right justified text.

With the reserved words L-R JUSTIFIED as the text type, the

justification is both left and right justified. At each breaking point of a line not caused by "#nl" and "#np", the spacing between each character is set so that the line ends at the right margin. "#nl" and "#np" breaking is handled in the same fashion as left justified.

The spacing between each line is dependent upon the character height specified in the text instruction (see below); therefore, in the case that the number of lines times the character height plus the line spacing is less than the Y dimension of the subplot area, the lower portion of the subplot area will remain empty. In other words, the spacing between each line will not be adjusted so that the complete subtask is used to draw the text.

If the character string is too long to be printed within the subplot area, only those characters that fit into the area will be printed. In the next section, it will be shown how a string identifier can be used to continue a character string from one subtask to the next.

If the text type is CENTERED, each line of the string is centered across the subplot area and evenly spaced so that the first line is at the top of the subplot area and the last is at the bottom. In all CENTERED text types, "#nl" and "#np" act only as breaking points of the lines. Breaking points must occur with one of these two symbol sets for CENTERED text.

For the TOP CENTERED text type, each line is centered across the subplot area starting at the top. Spacing between each line is treated in the same manner as justified text: it is independent of

the number of lines. For the BOTTOM CENTERED text type, the last line is printed just above the lower edge of the subplot area. Like TOP CENTERED, spacing between lines is treated in the same manner as justified text.

The text style is the type of character style to be used in drawing the character string. These may be any of the styles specified by DISSPLA (ref 4). At this point, only the default style will be used by inserting the reserved word SIMPLE. The other styles will be discussed in the next section.

The second parameter is the character string to be printed. Remember that all strings are enclosed in double quotations.

- Start is an integer value used as a pointer into the character string as the starting location. If the value is 1, the characters will be printed starting at the beginning of the string. This value cannot be greater than the length of the character string.

The length is an integer value or the reserved word CONTINUE used to determine the number of characters to be printed. If an integer value is used, only the number of characters specified will be printed. If the value is greater than the length of the string from the point specified by start, or if the reserved word CONTINUE is used, the string will be printed until either the end of string is reached or until the subplot area is filled. This eliminates the task of having to count the number of characters in a string.

One point should be made about using start and length. The starting and ending points in a string should not be within the locations where "#nl" and "#np" are found, since this will cause the system to bypass these commands.

Height is an integer or real value followed by the reserved word INCH or INCHES and specifies the character height to be used in printing the text.

Figure 35 and 36 show the use of left and right justified and top centered and centered text styles.

```

.
.
.

PAGE text_examp(VERTICAL,center,0,top)
SEGMENT l_just(0.00 INCH,4.50 INCH,
                3.25 INCH,9.00 INCH)
    FRAME
    TEXT(LEFT JUSTIFIED SIMPLE,
        "this is#nlan example#nl"&
        "of justified#nltext",
        1,CONTINUE,.50 INCHES)
END SEGMENT l_just
SEGMENT r_just(3.25 INCH,4.5 INCH,
                6.50 INCH,9.00 INCH)
    FRAME
    TEXT(RIGHT JUSTIFIED SIMPLE,
        "this is#nlan example#nl"&
        "of justified#nl text",
        1,CONTINUE,.50 INCHES)
END SEGMENT r_just
SEGMENT t_cent(0.00 INCH,0.00 INCH,
                3.25 INCH,4.5 INCH)
    FRAME
    TEXT(TOP CENTERED SIMPLE,
        "top#npcentered",
        1,100,.25 INCHES)
END SEGMENT t_cent
SEGMENT cent(3.25 INCH,0.00 INCH,
                6.50 INCH,4.50 INCH)
    FRAME
    TEXT(CENTERED SIMPLE,
        "this part not printed."&
        "centered#nltext",
        23,17,.25 INCHES)
END SEGMENT cent
END PAGE text_examp
.
.
.

```

FIGURE 35 Text Example

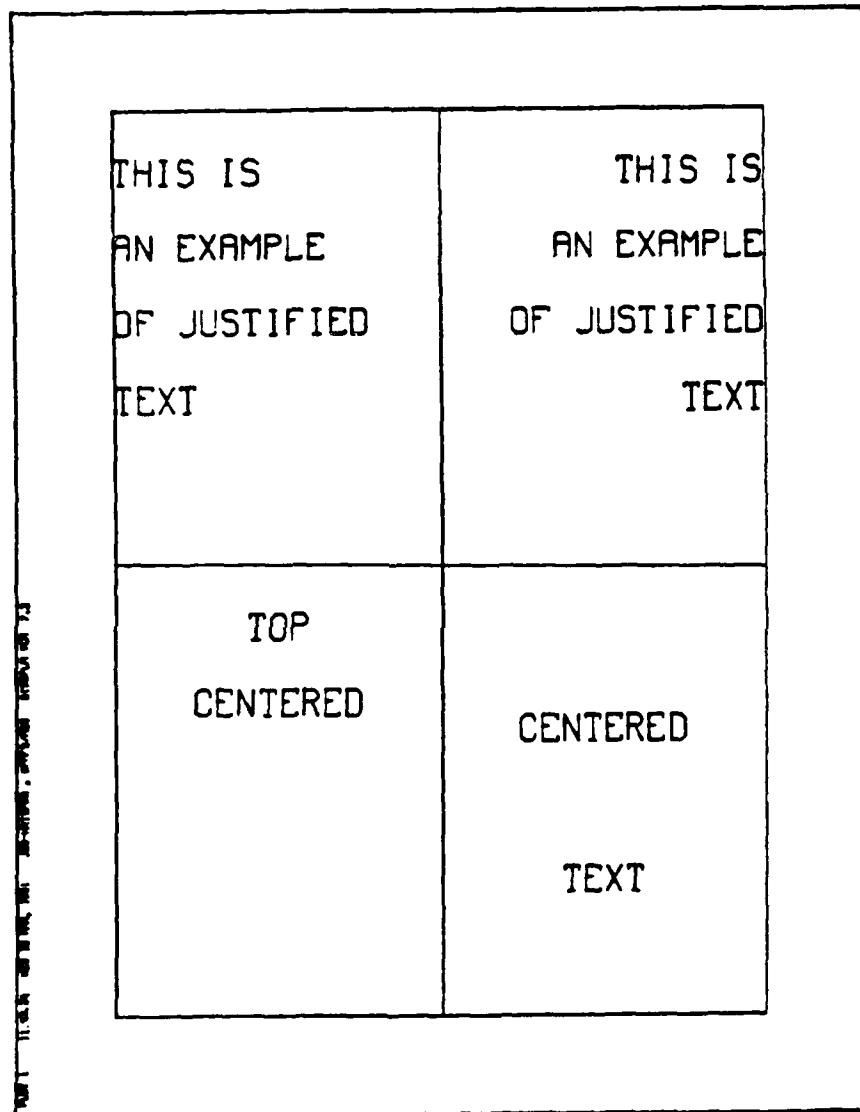


FIGURE 36 Text Example

II. PROGRAM ENHANCEMENT

Constants and Variables:

Up to this point, all programs have been shown to be written using literals as the basis of computation. The problem with this type of program implementation is that no matter how many times the program is executed, the results will always be the same. To change the output results, the user must edit the input program and change each of the literals. For programs that are even several page subtasks in length, this problem becomes almost impossible, since the user must understand how the changes will affect the output. To make programming more versatile, the programmer can assign a value or a set of values to a user defined identifier and then access these values through the identifier name. The same restrictions apply to value identifiers as those that apply to identifiers used in naming subtasks:

- 1) the length cannot exceed 10 characters,
- 2) the first character must be a letter,
- 3) only letters, digits, and the underscore are allowed,
- 4) an identifier cannot be a reserved word.

The first encounter of a value identifier is always within the declaration blocks of a program, where the identifier is declared to be either a CONSTANT or VARIABLE. As these names imply, once a constant is assigned a value (within the declaration block), the value cannot be changed. However, the values of variables can change during the execution of a program.

Declarations can occur only at the head of a subtask (PROGRAM,

SECTION, PAGE, or SEGMENT). An exception to this rule applies to the use of conditional statements. This topic will be discussed in section IV. A declaration block begins with the word DECLARE followed by the constant declaration set, the variable declaration set, and ends with the words END DECLARE. The constant declaration set begins with the word CONSTANT followed by a set of constant identifier assignments of the form:

identifier = value

The value may be integer, real, boolean, or character string.

The variable declaration set begins with the word VARIABLE followed by a set of variable declarations of the form:

- identifier list : type

an identifier list can be a list of from 1 to 10 identifiers separated by commas. The type denotes the type assignment to the identifier list.

It is not required that both constants and variables be declared within every declaration block. However, when both occur, the constant declaration set must occur before the variable declaration set.

An identifier can be assigned one of seven types: INTEGER, REAL, BOOLEAN, CHARACTER, STRING, AXIS, and UNIT. Once an identifier has been assigned a type, it cannot be changed. It is important to recognize the difference between assigning a type and assigning a value to a variable. The type specifies the type of value that the identifier will contain; therefore, while a variable may be reassigned different values, the type will always remain the same.

Since integers, reals, and booleans were discussed in the previous section, further description of these types will be deferred until the subsection on expressions. The use of the type CHARACTER for this version of the language is limited to logic comparisons only. At this point, it is enough to note that characters are defined to be character strings of length 1. For example, "5" represents the character value 5. This should not be confused with the integer value 5. The two are not equivalent.

When defining an identifier to be of type STRING, it is meant that the value the identifier will contain is a character string. The maximum length of the character string is defined by an integer value enclosed in parentheses following the word STRING in the declaration. Like the type assigned to a variable, the length of string types cannot change.

Axis variables, when declared, are assigned a record structure of values that are defined to be the five parts of an axis: the title, the type, the minimum and maximum points, the delta, and the ticks. Like the description previously stated for these parts, each axis variable record component has the same characteristics. The defining and handling of axis variables will be discussed later in the subsection on Assignment Instructions.

Unit variables are used as a method of scaling distances within a subplot area or as a conversion to any other system of measurement the user desires. As an example of scaling, the user could define the identifier MILES to be of type UNIT and then later, in an assignment instruction (see below), assign to it the value: 10

INCHES. At any point within the subtask after the assignment has been made, the user can refer to distances in units of MILES and the system will convert the scaling factor to INCH units (i.e. .5 MILES is converted to 5 INCHES). For example, the user could define the identifier CENTIMETER to be of type UNIT, assign the value (1/2.54) INCHES, and then refer to distance units in terms of centimeter. The user could then define MILLIMETER to be of type UNIT, and assign the value .1 CENTIMETER after the centimeter value assignment has been made.

An identifier of any of the four basic types INTEGER, REAL, BOOLEAN, and CHARACTER may also be declared as an array. An array is an ordered collection of values contained under a common identifier and consequently have the same type associated with that identifier. Arrays are used to store interrelated values together. A variable array declaration is of the form:

identifier list : ARRAY[index] OF basic type

where the basic type is one of the four reserved words mentioned above. The index is a list of one or more integer values that represent the sizes of the variable's dimensions. Each dimension value is called a subscript. At present, the system is limited to three dimensions.

```

1  PROGRAM ex_declare
2    DECLARE
3      CONSTANT pi=3.1415926
4    END DECLARE
5    :
6    SECTION sect_dec
7    DECLARE
8      VARIABLE
9        pager,index:INTEGER
10       flag:BOOLEAN /* test */
11       stack:ARRAY [2,2,10] OF REAL
12       xcharset, /* note how */
13       ycharset, /* each line */
14       xaxset, /* can be used */
15       yaxset: /* to define */
16       STRING (9)/* variables */
17    END DECLARE
18    :
19    PAGE page_dec
20    DECLARE
21      CONSTANT
22        setup="time axis"
23      VARIABLE
24        cm,mm:UNIT
25        axis_draw1,
26        axis_draw2:AXIS
27        index:REAL
28    END declare
29    :
30    END PAGEPAGE_DEC
31    :
32    END SECTION sect_dec
33  END PROGRAM ex_declare.

```

FIGURE 37 Constant and Variable Declarations

Figure 37 shows an example of how constants and variables are declared. In this example, each of the three subtasks are used to declare value identifiers. In the program subtask, only the constant PI is declared (lines 2-4). Since the value is real, the type assigned to the identifier PI is real. It is important to remember that the difference between declaring PI to be a constant with the

value 3.1415926, and declaring PI to be a real variable and assigning this value to it in the program, is that constants cannot be changed. In figure 37, the programmer may now access PI anywhere within the program without worrying that the value this identifier contains may not always be the standard value, as would be the case if PI were declared to be a variable. Thus, it can be seen that the use of constants should not be viewed as a restriction upon the user, but provides a failsafe against redefining a value identifier.

In the SECT DEC section subtask, only variables are declared (lines 13-23). In lines 18-22, it is shown that, while the structure is still syntactically correct (see the variable declaration form specified above), the format is unrestricted. The declaration on line 17 sets up a three-dimensional array of reals to be contained within the identifier STACK.

Within the PAGE_DEC page subtask, both a constant and variables have been declared. The identifier SETUP on line 34 is assigned a character string of length 9. Just as for integer or real constants, string constants may not be changed; consequently, the length also remains constant.

Before continuing to the description of arithmetic expressions, several important notes concerning the activation of constants and variables need to be discussed. Constants or variables become active at the point they are declared, and become inactive when the subtask that the constants and variables were declared within has been exited. In figure 37, the constant SETUP and the variables declared within PAGE_DEC, therefore, are local only to that

particular page subtask, and become inactive once line 48 has been executed. Constants and variables can only be accessed while they are active. The constant PI is a global constant since it is declared at the head of the program and remains active until the last line (64) is executed.

While not usually a good programming technique, the declaration of INDEX (line 15) demonstrates an extension to activation of a variable. Another variable INDEX was declared within PAGE_DEC (line 39). Within the page subtask, the SECT DEC INDEX becomes inactive and unaccessible during the execution of PAGE DEC and the value is unaffected while the PAGE_DEC INDEX is active. On completion of the page subtask (line 48), the PAGE_DEC INDEX becomes indefinitely inactive and the SECT_DEC INDEX is returned to the active state until the execution of line 63.

Arithmetic Expressions:

Expressions are used to compute results from a sequence of arithmetic operations. For this language, these operations are implicitly performed in the standard format for algebraic mathematics: left to right, proceeding according to specific rules of precedence. Table 3 defines the order or precedence that operations are performed. Lower order operations are performed first. In the case of a sequence of operations that occur at the same level (i.e. $-5+3-1$), the operations are performed left to right.

An operator is a word or symbol that defines the type of operation to be performed upon the operand(s). An operand will either

be a resulting value from a previous operation or one of the primaries listed in table 4. Tables 3 and 4 also show the restrictions on the type that each operand can be, and gives the resulting type for each operation. Table 5 defines the type symbols used in both tables.

Figure 38 illustrates the order of evaluation for an expression. Although each of the operators and operands are written in the same order, the use of the explicit evaluation "()" from table 4 shows the effect it has upon the precedence of evaluation. In the first case, the expression is completely implicit, therefore the expression becomes equivalent to:

$((10/5)-(2*(3^{**}2)))+1$

Explicit evaluations can be interpreted to mean: "If a left parenthesis is encountered at the start of an operand, evaluate what is within the parentheses and use this result as the operand of the operation." Whether the internal operation (within the parentheses) is a lower precedence than the external operation or not, the internal operation is always evaluated first.

In all three cases, each of the primaries are integer literals and each operation results in an integer value. In figure 39, the same expressions are used except that several of the literals are now reals to show that the value and type results vary from those in figure 38.

TABLE 3

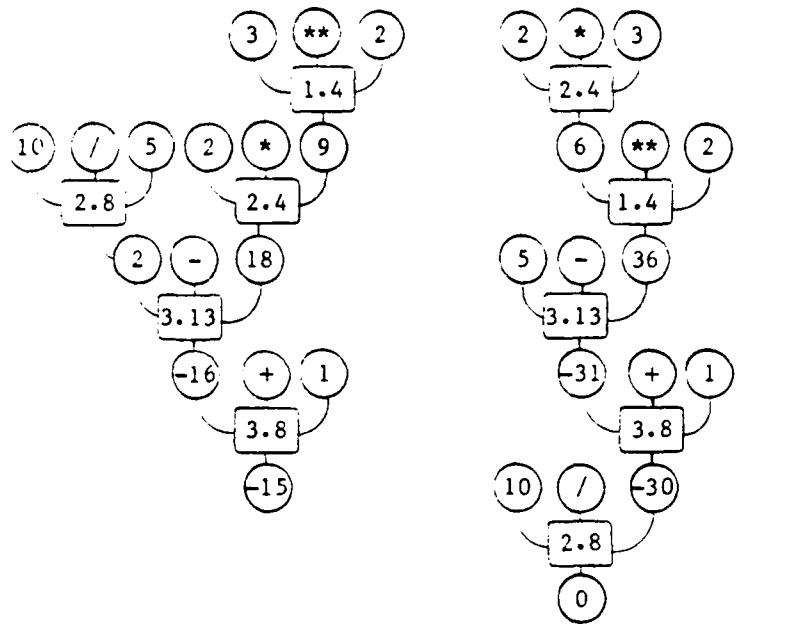
PRECEDENCE OF OPERATIONS					
LEVEL	OPERAND	OPERATOR	OPERAND	RESULT	OPERATION
1.1	R	**	R	R	exponentiation
.2	R	**	I	R	exponentiation
.3	I	**	R	R	exponentiation
.4	I	**	I	+I	exponentiation
2.1	R	*	R	R	multiplication
.2	R	*	I	R	multiplication
.3	I	*	R	R	multiplication
.4	I	*	I	I	multiplication
.5	R	/	R	R	division
.6	R	/	I	R	division
.7	I	/	R	R	division
.8	I	/	I	I	integer division
.9	I	MOD	I	I	modulus
.10	I	REM	I	I	remainder
3.1		+	I	I	unary plus
.2		+	R	R	unary plus
.3		-	I	I	unary minus
.4		-	R	R	unary minus
.5	R	+	R	R	addition
.6	R	+	I	R	addition
.7	I	+	R	R	addition
.8	I	+	I	I	addition
.9	CS	+	CS	CS	string concatenation
.10	R	-	R	R	subtraction
.11	R	-	I	R	subtraction
.12	I	-	R	R	subtraction
.13	I	-	I	I	subtraction
4.1	B	AND	B	B	conjunction
.2	B	OR	B	B	inclusive disjunc.
.3	B	XOR	B	B	exclusive disjunc.
5.1	BASIC	<	BASIC	B	compare l.t.
.2	BASIC	<=	BASIC	B	compare l.t.eq.
.3	BASIC	=	BASIC	B	compare eq.
.4	A	/=	A	B	compare not eq.
.5	BASIC	>=	BASIC	B	compare g.t.eq.
.6	BASIC	>	BASIC	B	compare g.t.

TABLE 4

EXPRESSION PRIMARIES			
OPERAND	PARAMETER(S)	RESULT	OPERATION
1 variable	[I list]	I/R/B/C	array variable
2 variable		I/R/B/C/S	
3 constant		I/R/S	
4 literals		I/R/B/CS	
5 (expression)		same	explicit evaluation
6 NOT	(B)	B	logic inverter
7 SIN	(R)	R	sine function(rad)
8 COS	(R)	R	cosine function(rad)
9 TAN	(R)	R	tangent function(rad)
10 INV SIN	(R)	R	inverse sine
11 INV COS	(R)	R	inverse cosine
12 INV TAN	(R)	R	inverse tangent
13 INTEGER	(R)	I	convert R to I
14 FLOAT	(I)	R	convert I to R
15 FRACTION	(R)	R	return fraction of R
16 ABSOLUTE	(I)	I	absolute value
17 ABSOLUTE	(R)	R	absolute value
18 POINT	(S)	+I	return pointer
19 LENGTH	(S)	+I	return string length
20 ORIGIN	(X)	+R	return X origin
21 ORIGIN	(Y)	+R	return Y origin
22 AREA	(X)	+R	return X length
23 AREA	(Y)	+R	return Y length
24 REFERENCE	(S , CS)	+I	return reference point
25 STRING	(I)	CS	return CS of I
26 STRING	(R)	CS	return CS of R

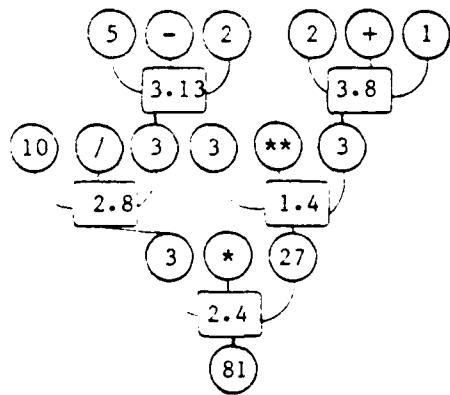
TABLE 5

TYPE SYMBOLS	
TYPE	
A	all type expressions excluding axis and unit variables
B	boolean expressions
BASIC	integer, real, boolean, or character expressions
I	integer expressions
+I	positive integer result
R	real expressions
+R	positive real result
CS	CHARACTER string
S	string identifier



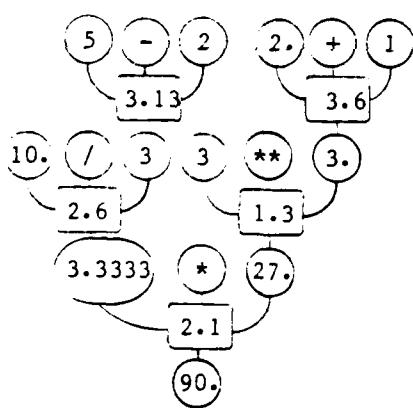
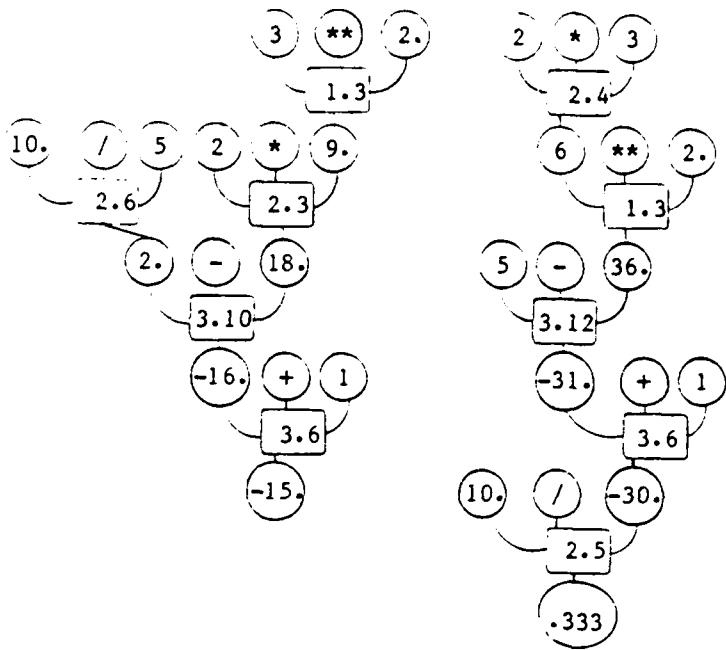
(a) $10/5-2*3^{**2}+1$

(b) $10 / (5 - ((2 * 3) ** 2 + 1))$



(c) $10/(5-2)*3^{2+1}$

FIGURE 38 Integer Expression Example



(c) $10./((5-(2*3)^{**}2.))+1$

FIGURE 39 Real Expression Examples

Functions:

In table 4, the primaries from line 6 to 26 are called functions. Functions are used to perform a specific evaluation on the parameter(s) passed. In most cases, the parameters may be any expression as long as the type resulting from the expression is of the type indicated under PARAMETER(S) in table 4. Exceptions of the use of expressions are where S denotes a string identifier, CS denotes a character string, and where X and Y are used (lines 20-23). Converts the result of an integer expression to type real. The value of the integer parameter is actually not changed, but the type is converted. For example, FLOAT(5) returns the value 5.0. FRACTION returns the fractional portion of a real expression result; therefore, a real expression may be broken into its integer and fraction by these two functions. ABSOLUTE returns the absolute value of a real or integer expression result, but does not affect the type (i.e. ABSOLUTE (5.3)=5.3 and ABSOLUTE(-10)=10).

The functions on lines 18-19 and 24-26 are string functions. POINT returns the pointer to the character in the string identifier immediately after the last character printed on the screen. This function is specifically used to interact with the text instruction when two or more pages or segments are needed to print a long string of text. If the string identifier has not been used within a text instruction, or if the complete character string has been printed, the value returned is 1. LENGTH returns the number of characters assigned to a string identifier. This value may be different than the length that the identifier was declared to be if the character string assigned is shorter. For example, if the identifier in the declaration:

`store_strn:STRING(25)`

is later assigned the character string "not long enough", then `LENGTH(store_strn)` will return the value 15, not 25.

The function `REFERENCE` is used to locate a character string within a string identifier by returning a pointer to the head of the character string `first` encountered in the identifier. In the example above, `REFERENCE (store_strn, "en")` returns the value 10. This function is often used with the `POINT` function to locate references to graphs within a character string so that these graphs can be plotted near their references. An example of the use of these functions will be given later. The `STRING` function is used to return the character string of an integer or real expression. This function can be used in concat concatenation of character strings (defined as character string addition). For example, "the temperature is" + `string(48.60)` + " degrees" produces the string: "the temperature is 48.6 degrees".

The `ORIGIN` and `AREA` functions provide a method of positioning page segments at particular points on the page. `ORIGIN(X)`, when used in a page subtask or in defining a segment subtask, returns the distance (in inches) in the X direction from the lower left physical corner of the page to the page physical origin. `ORIGIN(Y)` returns the distance in the Y direction. This function is used to define the lower left corner of a segment, when the position is to be independent of the page physical origin. For example, figure 40 illustrates how a segment may be positioned exactly (2.5 INCHES, 4.5 INCHES) from the lower left corner of the page no matter how the

margin and frame instructions affect the physical origin of the page. One problem still remains with this illustration. If the segment positions lie outside the page physical origin, errors will still occur. The ORIGIN function, when used within a segment subtask, will return the distance from the page physical origin from the segment physical origin. These values, in later versions of this language, will be used for user-defined windowing of objects within a page segment.

```
    .
    .
    .
PAGE seg_set_up(VERTICAL,center, . . .
MARGIN( . . .
FRAME(. . .
SEGMENT exact((2.5-ORIGIN(X)) INCHES,3 INCHES,
               (4.5-ORIGIN(Y)) INCHES, 3 INCHES)
    .
    .
    .
```

FIGURE 40 ORIGIN Function Example

AREA(X) returns the length of the subplot area in the X direction of the page or page segment, depending upon which subtask the function is called in. In the previous section it was stated that the position and size of each page subtask plot area must be calculated so that segment subplot points are not out of the page bounds. If the size of the page subplot area depends upon the evaluation of expressions, the dimensions may not be calculated until the program is executing. In figure 41, the page segments are defined according to this function. The page subtask is evenly divided into four equally-sized segments. Note that although the

margin and frame instructions affect the sizes of the segments, the expression values for the parameters of both cannot be evaluated previous to the execution of the page subtask procedure.

```
PROGRAM area_set
  DECLARE VARIABLE
    framsize,margsize:real
  END DECLARE
  .
  .
  .
  PAGE not_set(HORIZONTAL, left reset, . . .
  MARGIN(margsize)
  FRAME(framsize)
  SEGMENT set_up(0 INCH,(area(x)2) INCH,
  0 INCH,(area(y)2) INCH)
  .
  .
  .
  END SEGMENT setup_1
```

FIGURE 41 AREA Function example

Assignment Instructions:

Once a variable has been declared, and while the variable is active, it can be assigned a value. The value of an assignment must be of the same type as the type the variable was declared to be. There are no other restrictions upon an assignment instruction. An assignment may occur in one of two methods. The first type is an explicit assignment instruction called a set instruction. The formats for set instructions are as follows:

```
SET variable = expression
SET variable [array components] = expression
SET STRING string variable = expression
```

```
SET UNIT unit variable = expression    unit identifier  
SET AXIS axis variable = (title,type,min,max,delta,ticks)
```

The first is used for defining a variable of any of the basic types: INTEGER, REAL, BOOLEAN, and CHARACTER. The expression evaluated must be of the same type as the variable. For character variables, the expression may only be another character variable previously defined, a single character in double quotations, or the STRING function where the expression parameter is evaluated to be an integer between 0 and 9. No other restrictions apply. Expressions may be of any length.

The second set instruction is used for defining an array variable. The array components enclosed in brackets is a list of integer values that specify which component is being assigned. The number of integer values must be the same as the number of dimensions of the variable, and each integer value must be less than or equal to the corresponding simension in the declaration. All variables that have been declared as arrays must have specified array components in a set instruction.

The third instruction is used to define a string variable. The expression may be either another string variable or constant, a character string, the STRING function, or a concatenation of character strings and STRING functions. The length of the character string expression must not exceed the length of the string variable (determined in the declaration of the variable).

In the assignment of a unit variable, the expression must be of type INTEGER or REAL. The unit identifier may be either a previously defined unit variable or the word INCH or INCHEs.

For an axis variable assignment, all five record components may be defined as shown in the fifth set instruction format. Each component's characteristics are the same as those discussed in the previous section under the axis instruction, except that now expressions may be inserted. These characteristics are expanded here to include string variables and constants or concatenation of character strings and STRING functions for the title; integer or real expressions where the axis is not of type MONTH; and integer expressions for the delta and ticks components. These expansions may also be used in the components defined for the axis instruction too. They were deferred until the discussion on expressions.

- Any one of the axis variable's components may also be defined or redefined individually without having to declare all of the parts at once. These formats are:

```
SET AXIS axis variable.TITLE=title
SET AXIS axis variable.TYPE=type
SET AXIS axis variable.MIN=min
SET AXIS axis variable.MAX=max
SET AXIS axis variable.DELTA=delta
SET AXIS axis variable.TICKS=ticks
```

These set instructions provide an excellent way of defining each part at different locations in the program. The only restriction is that once the TYPE of an axis variable has been defined, it cannot be redefined to be another axis type unless all parts are defined at

once. This is required by the conversion necessary for MIN and MAX values, since their values depend upon the type of axis defined. Also, if MIN and/or MAX are defined to be a month (i.e. JAN, feb,...) prior to defining the type of axis, TYPE will automatically be set to MONTH.

The second type of assignment instruction involves retrieving data (data, constant values, and constants are used interchangably) from a specified file or from the input deck. Appendix B of this paper discusses how files are attached before the execution of the program and how the data may be retrieved. This appendix will describe only the formats of the input instruction and how the values retrieved are assigned to a variable. The type of data this system allows are: integer and real constants, boolean values (TRUE or FALSE), and character strings.

For retrieving data from an attached file, the input instruction format is:

INPUT identifier:TAPE number

where the number is an integer constant. Expressions may not be used in the place of this number. The current version of the system allows only input from TAPE1 and TAPE2. When the instruction is executed, values are retrieved left to right, line by line, ignoring any comments found. The identifier may be of one of the basic types or a string variable. If the variable has been declared to be an array, subscripts of the variable are not included. The system retrieves the number of values from the tape that correspond to the

dimensions of the variable. For arrays declared to be of more than one dimension, values are assigned as shown in table 6. As an example, `y_val_stk` is declared to be an `ARRAY[2,2,10] OF REAL`. Then in the instruction:

```
INPUT y_val_stk:TAPE 1
```

the next 40 (2 times 2 times 10) data values are retrieved from TAPE1: the first value is assigned to `y_val_stk[1,1,1]`, the second to `y_val_stk[1,1,2]`, and continuing until the last value is assigned to `y_val_stk[2,2,10]`.

TABLE 6

Array Variable Assignments	
<u>VALUE READ</u>	<u>ASSIGNMENT</u>
1st	[1 , 1 , 1]
2nd	[1 , 1 , 2]
:	:
Nth	[1 , 1 , N]
N+1th	[1 , 2 , 1]
:	:
2Nth	[1 , 2 , N]
2N+1th	[1 , 3 , 1]
:	:
MNth	[1 , M , N]
MN+1th	[2 , 1 , 1]
:	:
LMNth	[L , M , N]

The format for retrieving data from the input deck is:

```
INPUT identifier:TERMINAL
```

The word TERMINAL is used as the source locator, instead of for

example DECK, because the language was originally designed to run interactively with the user. This simply means that the user could choose the values for variables during runtime of the program by typing them in from a terminal as the input instructions are executed. The identifier is handled in the same manner as the tape input instruction. In both of these two instructions, as data is retrieved, the datum types are individually compared with the variable identifier type before assigning the value. At any time, if the types differ, an error message is printed and execution of the instruction is stopped.

Data may also be assigned directly to a variable identifier by the format:

INPUT identifier: DATA / data list /

where the data list is a list of integer or real constants, a list of boolean values (TRUE or FALSE), or a character string. As was noted in the previous section in the description of the GRAPH instruction, it is possible to reduce a list of constants that are the same to the form:

value : number

to prevent having to type a long list of consecutive values. The number must be an integer constant and represents the number of times the constant is to be repeated.

For instance, if FLAGS is declared to be an ARRAY[100] OF

BOOLEAN and the user wants to initialize the first 50 to TRUE and the next 50 to FALSE, the input instruction could be:

```
INPUT flags:DATA / TRUE:50,false:50 /
```

Although the language allows assigning a character string to a string identifier with this instruction, it should be rarely used if at all, since

```
SET STRING string identifier = character string
```

is equivalent to the instruction

```
INPUT string identifier:DATA / character string /
```

and since the data list in the input instruction does not allow character string concatenation or the use of the STRING function. In the same light, assigning a basic type variable not defined as an array is simpler and easier to understand with the SET instruction since the data list does not allow expressions either.

Besides the normal type of output that this language provides to a graphics terminal, a specific instruction is available to output data to a specified tape or to the output file. The formats of the output instruction are similar to those of the input instruction:

```
OUTPUT identifier:TAPE number
```

```
OUTPUT identifier:TERMINAL
```

The system has two output tapes that data may be written to: 3 and 4. The identifier may be an array variable, basic variable, string variable, an axis variable, or any of the constant identifier types. Outputting string identifiers or variables to the output file (TERMINAL), used with the trace function of the system, provides a method of debugging any program. When the identifier is an array variable, the values are written in the same sequence as they are read in the input instruction, shown in table 6.

Page Format Instructions:

At this point, before describing the page format instruction, it would be useful to define a term that will be used frequently from this point on. A UNIT VALUE is an integer or real representation of a specified distance. The distance value may be of any integer or real type expression. This expression is followed by a variable of type UNIT or the unit word INCH or INCHES. Therefore, a unit set instruction can be represented in the form:

SET UNIT unit variable = unit value

as well as defining the segment parameter list as of the form:

SEGMENT segment identifier(unit value, unit value,
unit value, unit value)

the syntax diagrams in appendix A show all locations where unit values are used.

There are three page format instructions available to the user for redefining the size, the grace margin, and the binding margin of pages. The values in the previous section of this appendix describing these formats are default values of the system. Since these instructions affect the output plots, they can only be called in the main program or in a section subtask, not within a page or page segment.

To redefine the physical border or size of a page, the format is:

BORDER = unit value BY unit value

It is unimportant whether the length or height of the page is specified first. If a page is defined to be VERTICAL, the longer side is always the vertical dimension of the page and the shorter is always the horizontal dimension. For a HORIZONTAL page, the format is reversed: the longer side is the horizontal dimension. For pages that are square (i.e. BORDER = 5 CENTIMETERS BY 5 CENTIMETERS), HORIZONTAL and VERTICAL pages are the same.

The grace margin of a page is redefined by the grace margin instruction:

GRACE = unit value

This instruction also implicitly affects the location page numbers are printed. If the page number location is TOP, the number

will be half the distance of the grace margin from the top edge of the page border; for BOTTOM, the number is printed half the distance from the bottom edge. If the grace margin is set to 0 inches, or if the margin is defined to be less than the character height (see SECTION III of this appendix), the page number will not be printed.

The third page format instruction redefines the binding margin of a page.

BINNING = unit value

If the binding margin is set equal to the grace margin, the user should note that the page margin parameter in the page construct list becomes ambiguous since LEFT RESET, RIGHT RESET, and CENTER produce the same page margins: centered within the page.

An important point should be made about using these format instructions. Like constants and variables declared in a specific task, the page format instructions are active only in the subtasks that they are called. For example, if a binding margin instruction is called within the main program (before a section or page subtask), then the binding format is global to the complete program and affects every page subtask. However, if called within a section, the format remains local to that subtask. If a page format instruction is used in both the main program and within a section, the format value defined within the main program becomes inactive during the execution of the section, but is reactivated at the end of the section.

Page Margin Defaults:

In the previous section of this appendix, it was shown that the binding margin is defined with the words LEFT RESET, RIGHT RESET, or CENTER as the second parameter of the page construct list. In most cases, however, when text and graphs are bound, pages are printed both on the front and back. To prevent having to insert the LEFT and RIGHT RESETs in every page parameter, a default allows consecutive pages to be printed with opposing binding margins. The default is activated when a blank space is found as the parameter.

Initially, upon entering the first page of the program or a section subtask, the default is equivalent to LEFT RESET, where the binding margin is placed on the left side of the page. In the second page subtask, the default is a right binding margin. At any time, the default is reset when LEFT RESET or RIGHT RESET is inserted as the parameter. When CENTERed page margins are used within a sequence of binding pages, the default is affected as if a blank was inserted. (That is, if a left binding margin page is followed by a CENTERed page, the default for the next page will be a left binding margin).

Graph Instructions:

In the previous section, the format of the graph instruction, along with the preparation instructions, was described for drawing simple linear, bar, and pie graphs. ASGOL provides several variations of these interpolations. Two smoothing techniques provided by DISSPLA are available through ASGOL. SPLINE is a cubic

spline interpolation that fits third order polynomials between each plotting point on the graph. This type of interpolation produces the smoothest fit for non-irregular data; however, in fitting the curve through each point, it also tends to produce oscillations. Also, the total number of points per curve that may be interpolated is limited to 102 (100 plus the end points). SMOOTH, similar to SPLINE, is a spline smoothing technique that is used for data that is somewhat scattered and when the SPLINE method does not produce a sufficient smoothing effect. SMOOTH produces the smoothest possible curve that passes, on the average, within a specific distance of the data points. These two techniques are described fully in the DISSPLA manual (ref 5).

ASGOL also has several variations of the basic BAR interpolation. STEP interpolation is a degenerate cousin of the bar. In STEP graphs, horizontal lines are drawn through each plotting point of each curve, ending halfway between the X coordinates of the plot. Vertical lines are then drawn connecting these end points and producing a "stepping" effect. Graphs are not shaded with the STEP interpolation. STACK BAR provides a method of showing "sums of values" at each X coordinate of the graph by stacking successive bar plots, instead of placing them side by side as is the method of the basic BAR interpolation.

In addition to the direct input of Y coordinate values discussed in the previous section, ASGOL also allows values to be passed with the use of an array integer or real variable. The format for the graph instruction is not different; the direct input parameter is simply replaced by the variable name:

GRAPH title (graph type, number of plots, variable)

Using the variable method is a much "cleaner" way of plotting points on a graph, since the instruction is not cluttered with a long list of values.

For plotting a single curve or a single bar plot on a graph (number of plots=1), the variable can be declared to have 1, 2, or 3 dimension subscripts. For a single dimensional array, the subscripts must be greater than or equal to the number of points to be plotted (defined by the delta of the X axis). For 2 and 3-dimensional arrays, if the number of points specified by the X axis is N, the Y values used for plotting are the array components [1,1] through [1,N] and [1,1,1] through [1,1,N] respectively.

For plotting multiple curves or bar plots on a graph, if the number of plots = M (M>1), then the variable must be declared to be a 2- or 3-dimensional array, where the minimum dimensions in the declaration are [M,N] and [1,M,N] respectively. Table 2 can now be revised to illustrate how the Y values of each plotting point are obtained from a 3-dimensional array, shown in table 7. For 2-dimensional arrays, the first subscript (1) is removed.

TABLE 7

X COORDINATE OF ARRAY COMPONENTS	
COMPONENT	X coordinate
[1,1,1]	minimum,1st plot
[1,1,2]	1st delta,1st plot
:	:
[1,1,N]	maximum,1st plot
[1,2,1]	minimum,2nd plot
:	:
[1,2,N]	maximum,2nd plot
[1,3,1]	minimum,3rd plot
:	:
[1,M,N]	maximum,Mth plot

In this table, it can be seen that if the variable's second dimension was declared to be J, where $J > M$, and the third dimension was declared to be K, where $K > N$, then the values contained in the variable's subscripts [1,M+1,1] through [1,M+1,K], . . . , [1,J,1] through [1,J,K] become inaccessible for plotting.

One of the most powerful tools that ASGOL, version 1, provides is the stack selection for graphs. With this feature, it is possible to draw a series of linear or bar graphs, stacked vertically within the subplot area. The only restrictions that apply to this option is that no more than 6 graphs may be stacked, all graphs must be of the same interpolation type, and each graph must contain the same number of curves or bar plots. The stack option is useful when too many plots on a single graph tend to "clutter" together, making it difficult to distinguish individual curves. Each consecutive graph is drawn directly above the previous graph, and for each graph, the Y

axis is redrawn. The X axis is drawn only once, along the lower horizontal edge of the bottom graph.

When the legend and grid instructions are used as preparation instructions for stacked graphs, the instructions are called once for each graph to be drawn (i.e. For a stack of 5 graphs, there would be 5 legend instructions). The first legend or grid instruction is used by the bottom graph, the second by the graph above the bottom graph, etc. If more legend or grid instructions than the number of graphs are called, the remaining instructions will be ignored. If less instructions are called, the remaining graphs will be drawn without a legend or grid.

- The format for the stack graph is:

```
GRAPH title (STACK OF number graph type,  
                  number of plots, variable)
```

where number is an integer constant between 1 and 6. If the optional title is used, the character string will be printed once at the top of the subplot area.

Although it is possible to input the Y values directly for stacked graphs, it is not recommended, since the X coordinate locations for the plotting points become much more difficult to determine. However, with an array variable, the extensions of the stack option are simple and straight forward. If the number of stacked graphs=L, then the variable must be a 3-dimensional array where the minimum dimensions in the declaration is [L,m,n]. M and N

denote the number of plots and number of points per plot as mentioned above. Where the number of plots per graph is one (single curve or bar plot), the variable may be a 2 or 3-dimensional array with minimum dimensions $[L,n]$ and $[1,L,n]$ respectively. Table 8 illustrates the expansion of the 3-dimensional array in table 7 for stacked graphs. In extending table 7, if the first dimension of the variable is declared to be I , where $I>L$, and the second and third dimensions are J and K , $J>M$, $K>N$, as mentioned above, then the values contained in the variable's subscripts $[L+1,1,1]$ through $[L+1,J,K]$, . . . , $[I,1,1]$ through $[I,j,k]$ become inaccessible for plotting.

TABLE 8

X COORDINATES FOR STACKED GRAPHS	
COMPONENT	X COORDINATE
$[1,1,1]$	minimum,1st plot,1st graph
$[1,1,2]$	1st delta,1st plot,1st graph
:	:
$[1,M,n]$	maximum,Mth plot,1st graph
$[2,1,1]$	minimum,1st plot,2nd graph
$[2,1,2]$	1st delta,1st plot,1st graph
:	:
$[2,M,n]$	maximum,Mth plot,2nd graph
:	:
$[L,1,1]$	minimum,1st plot,Lth graph
:	:
$[L,m,n]$	maximum,Mth plot,Lth graph

An important note should be made about the use of the stack option. Since all graphs are form-fitted to the subtask's subplot area, the user should make sure that the vertical dimension of the area is large enough that the graphs do not get "crunched". the

system will not write outside of the subplot area or chop off graphs to make the first few recognizable. These are left for the user's discretion.

Text Instruction:

In printing long character strings in text form on the screen, it was stated in the previous section that it is possible to continue the string in consecutive page or segment subtasks when the subplot area is not large enough for the complete character string. This variation of the text instruction does not involve any preparation-instructions nor any changes to the actual format of the instruction.

The format is written here for convenience.

```
TEXT (text type text style, string identifier,  
      start, length, height)
```

The changes in this instruction from the previous one is made in the second parameter, where a string constant or variable replaces the actual character string.

Each string constant and variable, when declared, is assigned a pointer value used specifically by the text instruction. When a string constant is assigned a character string within the declaration block, and when a string variable is assigned a character string with the SET or INPUT instruction, this pointer is initialized to point at the first character. The pointer is then updated each time the text

instruction does an operation on the variable or constant. If the text instruction encounters the end of the character string while the characters are printed, the pointer is reset to point again to the head of the string, and stops execution. If, however, the text instruction reaches the end of the subplot area before the complete character string is printed (or the number of characters in the length parameter is reached), the pointer is updated to point to the character after the last character drawn on the screen. The POINT function described earlier is used to access this pointer value for determining the location the text instruction is to begin printing.

If the word NEXT is inserted as the start parameter for the text-instruction, the instruction begins printing characters starting at the character that the pointer specifies. NEXT can be used as the parameter for the first text instruction call after the string identifier has been assigned, in which case the instruction starts with the first character of the string.

ASGOL does not provide a method of explicitly setting the pointer of a string identifier to a specific location since the start parameter may also be an integer expression; however, in rare cases, the pointer may be reset implicitly to point to the head of the string with the SET instruction:

```
SET STRING string variable = string variable
```

where both string variables are the name of the string identifier. String constants cannot be reset due to the condition

that constants cannot be reassigned. If this instruction is used, it is recommended that the programmer provide comments in the program describing why this instruction is included.

The text instruction also uses a default value for the character height of the text to be printed if a blank space is encountered as the parameter. The default height is 0.14 inches or the current character height specified in the program (see Section III, String Manipulation).

III. STRING MANIPULATION:

Since ASGOL was designed with maximum capability for DISSPLA -string manipulation, it would be impossible to describe completely in a few pages each of the facets DISSPLA provides. Therefore, the user is referred to the DISSPLA manual version 8.2 in the subsequent section for further description of string manipulation for this language. It is advised that the user study specifically chapters 6, 22, and 24 of Part B of that manual to become familiar with the character styles and fonts DISSPLA provides and how they are manipulated. However, the user is relieved of having to understand the FORTRAN call instructions since ASGOL provides this construction in its language.

In the definition of a character string in the first section of this appendix, it was stated that the symbols (,), +, -, *, and / (defined as shift characters) are used by the system for character font manipulation. Since DISSPLA allows up to six character sets to be active at any time, the system uses the six most common fonts, as

shown in table 9. When a character string is printed, if the first character is not a shiftcharacter, the LOWER case ROMAN font is used until a shift character is detected or until the end of string is reached. When a shift character is detected, the font remains active until another shift character is found.

TABLE 9

SHIFT CHARACTER REPRESENTATION	
<u>CHARACTER</u>	<u>FONT</u>
(UPPER ROMAN
)	LOWER ROMAN
+	UPPER SCRIPT
-	LOWER SCRIPT
*	UPPER ITALIC
/	LOWER ITALIC

DISSPLA, however, has eight other character sets available: UPPER and LOWER case RUSSIAN, UPPER and LOWER case GREEK, HEBREW, SPECIAL, MATH, and INSTRUCTION. Any of these character fonts can become active simply by replacing the character font representation for one of the six shift characters. Any or all of the implicit fonts can be changed. The format of the change instruction is:

CHANGE implicit font TO explicit font

where the implicit font is ROMAN, SCRIPT, or ITALIC (preceeded by the word UPPER or LOWER to specify the case) and the explicit font is GREEK or ROMAN (preceeded by UPPER or LOWER), HEBREW, SPECIAL,

MATH, or INSTRUCTION. INSTRUCTION is a special case in that it is not a character font, but acts upon the succeeding characters similar to the fonts. INSTRUCTION is discussed separately below.

For example, the LOWER case SCRIPT shift character - could be changed to represent the MATH font by the instruction:

CHANGE LOWER SCRIPT TO MATH

After the execution of this instruction, each time the shift character - is detected, the MATH font becomes active.

At any point of execution, no more than one shift character can be assigned to a character font. If, for example,

CHANGE UPPER ITALIC TO UPPER GREEK

was later followed by the instruction

CHANGE LOWER ROMAN TO UPPER GREEK

within the same subtask, an error would result because the program has tried to assign both shift characters * and) to the UPPER case GREEK font.

The INSTRUCTION option affects the position and properties of the characters to be drawn following the shift character of the font to be used. Prior to this shift character, the shift character representing the INSTRUCTION option and the instructions that act

upon the text are inserted. Instructions consist of single character commands followed by its arguments if any. At the end of the character string that the instructions act upon, the instructions are inserted again with 'x' as its argument. This resets the instructions previously activated. None of the instruction symbols or shift characters are printed to the output. The DISSPLA manual (ref 5) has the complete list of instruction commands.

The INSTRUCTION option provides a method of drawing characters, affecting the character height, angle, style, and font within the character string instead of as a parameter to an instruction. However, a precaution should be made when using this option. Commands that affect the properties of the character string (i.e. Character height) will be invisible to someone reading the program if the person is unfamiliar with the instruction commands or if the string is read from an input tape or file. Users should document their programs to include how these commands are used to produce the output.

The height of character strings can be changed by the height instruction and can occur in any task:

HEIGHT = unit value

By default, characters are printed with a height of 0.14 inches. When the height instruction is executed, the height of all characters printed on the screen is affected, except those printed with the text instruction without the default option for character height.

Like the page format commands, both the change and height instructions are local to the subtask that the instructions are called. These instructions may occur in any subtask, and can occur any number of times. The user should also realize that when the change instruction is used, it affects all of the strings drawn on the screen. One possible mistake that could be made is changing the font representation for one of the shift characters between two page subtasks that are using the text instruction to draw a continuing character string to the screen.

TABLE 10

Reserved Words and
Special Symbols

absolute	false	log	scmplx
all	feb	logarithmic	script
and	float	lower	section
apr	for	l-r	segment
area	fraction	mar	sep
array	frame	margin	set
arrow	framed	math	simple
aug	gothic	max	simplx
axis	grace	may	sin
bar	graph	min	smooth
binding	greek	mod	special
boolean	grid	month	spline
border	hebrew	next	stack
bottom	height	not	stacked
by	help	nov	step
cartog	horizontal	oct	string
case	if	of	tan
-center	inch	or	tape
centered	inches	origin	terminal
change	input	others	text
character	inside	output	then
complx	instruction	outside	ticks
constant	integer	page	title
continue	inv	pie	to
cos	italic	point	top
create	jan	program	triplx
data	jul	real	true
dec	jun	reference	type
declare	justified	rem	unit
delta	left	repeat	until
do	legend	reset	upper
down	length	right	variable
draw	line	roman	vertical
duplx	linear	run	while
else	list	russian	x
end		scan	xor
			y
:	+	*	/
**	/=	()
=	,	.	[
}	<	<=	>
>=			

VITA

James D. Hart was born on 4 September 1955 in Salina, Kansas. He graduated from Sebring High School, Sebring, Florida, in June 1973 and attended South Florida Junior College for two years. After receiving an Associate of Arts degree, he attended the Florida Technological University (now the University of Central Florida) in Orlando, Florida for his Bachelor of Science in Engineering degree Engineering Mathematiiss and Computer Systems (EMCS). Upon graduation in August 1979, he received a commission in the United States Air Force through the ROTC program where he entered the School of Engineering, Air Force Institute of Technology, for graduate studies in Computer Science.

Permanent Address: Rt. 1 Box 143DF
Crestview, FL. 32536

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/GCS/MA/81M-2	2. GOVT ACCESSION NO AD 4100 814	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) ASGOL- AN ALGOL-STRUCTURED GRAPHICS ORIENTED LANGUAGE	5. TYPE OF REPORT & PERIOD COVERED M.S. THESIS	
7. AUTHOR/s) James David Hart, 2nd Lt., USAF	6. PERFORMING ORG. REPORT NUMBER	
9. PERFORMING ORGANIZATION NAME AND ADDRESS Air Force Institute of Technology Wright-Patterson Air Force Base, Ohio 45433	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS	
11. CONTROLLING OFFICE NAME AND ADDRESS Air Force Flight Dynamics Laboratory Wright-Patterson Air Force Base, Ohio 45433	12. REPORT DATE March, 1981	
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	13. NUMBER OF PAGES	
16. DISTRIBUTION STATEMENT (of this Report)	15. SECURITY CLASS. (of this report)	
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)	15a. DECLASSIFICATION DOWNGRADING SCHEDULE	
18. SUPPLEMENTARY NOTES APPROVED FOR PUBLIC RELEASE UNDER E.O. 13016 Director of Public Affairs 21 MAY 1981	FREDERIC C. LYNCH FREDERIC C. LYNCH, Major, USAF Director of Public Affairs	
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Graphics Language LR(1) Parsing System Block Structured Language DISSPLA Software Package		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) An ALGOL-like graphics oriented language and system was designed to provide a block-structured format to graphics programs. The system was developed with an LR(1) parsing procedure technique, and graphs are constructed using the DISSPLA software package as the "host" to generate device-independent plot files. The language produces linear, bar and pie graphs, as well as having a text processor to draw a variety of character styles and fonts for documentation.		

SECURITY CLASSIFICATION OF THIS PAGE/When Data Entered

SECURITY CLASSIFICATION OF THIS PAGE/When Data Entered

END

DATE
FILMED

7-18-81

DTIC